

CodeWarrior®

Targeting Win32



Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

Revised: 980829 ACD

Metrowerks CodeWarrior copyright ©1993–1998 by Metrowerks Inc. and its licensors.
All rights reserved.

Documentation stored on the compact disk(s) may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from Metrowerks Inc.

Metrowerks, the Metrowerks logo, CodeWarrior, and Software at Work are registered trademarks of Metrowerks Inc. PowerPlant and PowerPlant Constructor are trademarks of Metrowerks Inc.

All other trademarks and registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISK(S) ARE SUBJECT TO THE LICENSE AGREEMENT IN THE CD BOOKLET.

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation 9801 Metric, Suite #100 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Ordering	Voice: (800) 377–5416 Fax: (512) 873–4901
World Wide Web	http://www.metrowerks.com
Registration information	register@metrowerks.com
Technical support	support@metrowerks.com
Sales, marketing, & licensing	sales@metrowerks.com
CompuServe	Goto: Metrowerks

Table of Contents

Table of Contents	3
1 Introduction	9
Metrowerks Year 2000 Compliance	9
Read the Release Notes!	10
What's New in This Release.	10
External Editor Support	10
Support For Multiple Document Interface	11
Online Help Support	11
Support For AMD™ 3DNow!™	11
New Command-line Arguments	11
CodeWarrior and Its Documentation	11
What's in This Manual	13
Where to Go from Here.	14
2 Getting Started	17
System Requirements	17
Installing CodeWarrior for Win32/x86	18
CodeWarrior Compiler Architecture	18
Development Tools for Win32/x86	19
CodeWarrior IDE.	19
CodeWarrior Debugger for Win32/x86	20
CodeWarrior C/C++ Compiler for Win32/x86	21
CodeWarrior Pascal Compiler for Win32/x86	21
Resource Compiler for Win32	22
CodeWarrior Linker for Win32/x86.	22
Metrowerks Standard Libraries.	23
Win32 API Headers.	23
MFC Application Framework	24
Microsoft Compatibility	24
The Development Process with CodeWarrior	25
3 CodeWarrior for Win32: A Tutorial	29
Tutorial: Meet Ray the Tracer	29

Table of Contents

Phase 1: Creating a Project	30
A Look at the Code	38
Phase 2: Getting Resourceful	39
Phase 3: Handling Commands.	41
Phase 4: What, Pray Tell, Is a DLL?.	44
Phase 5: All Together Now	47
4 Creating Win32/x86 Applications	51
Creating an Application	51
5 Creating Static Libraries for Win32/x86	63
About Static Libraries	63
Creating a Static Library	64
6 Creating Win32/x86 Dynamic Link Libraries	65
About Dynamic Link Libraries	65
A Primer on Exported Names	66
Exporting Names.	66
Importing Names	68
Name Transformations	68
DLL Initialization	71
Creating a Dynamic Link Library	73
7 Creating Win32 Resources	75
Including Resource Files in a Project	75
Win32 Resource Compiler	76
8 Target Settings for Win32/x86	79
Target Settings Overview	79
Settings Panels for Win32/x86.	81
Target Settings	82
x86 Target	84
Windows RC	86
x86 Processor	87
x86 Linker	92
Runtime Settings	95

	Debugger Settings	97
	x86 Exceptions	98
9	C and C++ for Win32/x86	101
	Number Formats for Win32/x86	102
	Win32/x86 Integer Formats	103
	Win32/x86 Floating-Point Formats	104
	Calling Conventions for Win32/x86	104
	Variable Allocation for Win32/x86	106
	Register Variables for Win32/x86	106
	Optimizing Code for Win32/x86.	107
	Register Coloring.	107
	Instruction Scheduling	108
	Linker Issues for Win32/x86	108
	Deadstripping Unused Code and Data	109
	Link Order	109
10	Pascal for Win32/x86	111
	Language Extensions for Win32/x86	112
	Turbo Pascal Support	112
	Number Formats for Win32/x86.	113
	Win32/x86 Integer Formats	113
	Win32/x86 Floating-Point Formats	115
	Calling Conventions for Win32/x86	116
	CodeWarrior Default Calling Convention	117
	Microsoft-Standard Conventions	117
	__cdecl calling convention.	118
	__fastcall calling convention	118
	__stdcall Calling Convention.	119
	The DLLEXPORT and DLLIMPORT Modifiers	120
11	Inline x86 Assembler	123
	Working With Inline Assembly for x86	123
	Inline Assembler Syntax for x86	124
	Supported Directives in Inline Assembly	125
	Labels Supported.	125

Table of Contents

Using Local Variables and Arguments in x86 Assembly	126
12 Debugging for Win32/x86	127
Project Settings for Debugging.	127
Debugging an Application	128
Debugging a DLL	128
Debugger Preference Panels.	129
Runtime Settings	129
Setting Up for Remote Debugging	130
Connecting the Computers	131
Configuring the Connection Software	132
13 Libraries and Runtime Code for Win32/x86	135
Standard Libraries for Win32/x86 Overview	135
The Metrowerks Standard Libraries	135
File and Memory Restrictions	137
Exception Handling in MSL C++	137
Libraries for Win32 Development	138
The Microsoft Foundation Classes	139
Win32-Specific Functions in MSL C	139
14 Optimizing Code for AMD-K6®	141
Optimizing for AMD-K6® Overview	141
3DNow!™ Optimizations.	141
Calling Conventions	141
Minimizing Mode Switching.	142
Vectorization	143
Pragma Support	146
Library Considerations	147
15 Troubleshooting	149
Troubleshooting Overview	149
Win32 Programming Problems	149
Errors With min() and max()	150
Compiling a Project for Mac OS on Windows NT	150

Table of Contents

Index	153
--------------	------------

Table of Contents



Introduction

This manual describes how to use CodeWarrior to develop code targeted at the Win32®/x86 platform, including stand-alone application programs, dynamic link libraries (DLLs), and static libraries. The manual also shows how to set Win32/x86 project options and create Win32 resources, describes CodeWarrior's Win32/x86-specific run-time libraries, and tells how to debug Win32/x86 code with the IDE debugger.

Win32 programs created with CodeWarrior will run on any x86-based system supporting the Win32 interface, including the following specific versions of the Microsoft Windows® operating system:

- Windows 98
- Windows 95
- Windows NT

This introduction includes the following sections:

- [Metrowerks Year 2000 Compliance](#)
- [Read the Release Notes!](#)
- [What's New in This Release](#)
- [CodeWarrior and Its Documentation](#)
- [What's in This Manual](#)
- [Where to Go from Here](#)

Metrowerks Year 2000 Compliance

The Products provided by Metrowerks under the License agreement process dates only to the extent that the Products use date data provided by the host or target operating system for date representations used in internal processes, such as file modifications. Any Year 2000 Compliance issues resulting from the operation of the Products

Introduction

Read the Release Notes!

are therefore necessarily subject to the Year 2000 Compliance of the relevant host or target operating system. Metrowerks directs you to the relevant statements of Microsoft Corporation, Sun Microsystems, Inc., Apple Computer, Inc., and other host or target operating systems relating to the Year 2000 Compliance of their operating systems. Except as expressly described above, the Products, in themselves, do not process date data and therefore do not implicate Year 2000 Compliance issues.

For additional information, visit: <http://www.metrowerks.com/about/y2k.html>.

Read the Release Notes!

Before you use the CodeWarrior IDE or a particular tool, you should read the release notes. They contain important last-minute information about new features, bug fixes, and incompatibilities that *may not be included in the documentation*.

The release notes directory is always included as part of a standard CodeWarrior installation. The release notes directory is also located at the top level of the CodeWarrior CD.

What's New in This Release

CodeWarrior's IDE has many new features. The following are most notable:

External Editor Support

The Windows IDE can now launch an external editor whenever the IDE needs to display a text file. For more information, refer to the "Configuring IDE Options" chapter of the *IDE User Guide*.

Support For Multiple Document Interface

By default, the IDE now uses the standard Multiple Document Interface (MDI). Refer to the *IDE User Guide* for more information on this options and others on the **IDE Extras** preference panel.

Online Help Support

The new online documentation has “How do I...” task-oriented support topics and easier navigation with improved viewers. To learn more about this new feature, refer to the *IDE User Guide*.

Support For AMD™ 3DNow!™

CodeWarrior now supports Applied Micro Devices 3DNow!™ Technology. Refer to [“Optimizing Code for AMD-K6®” on page 141](#) for more information about CodeWarrior’s support for this technology.

New Command-line Arguments

CodeWarrior now has new command-line arguments for building projects that allow scripted builds and control by external applications. For a list of the command-line switches and their descriptions and error codes, read the `CW IDE Windows command line.txt` file on the CodeWarrior CD.

CodeWarrior and Its Documentation

CodeWarrior is a multi-host, multi-language, multi-target development environment. What does that mean?

Multiple hosts CodeWarrior runs on several different operating systems, including Windows and Mac OS. The features, human interface, and operation of CodeWarrior is very similar on all hosts.

Multiple languages You can use CodeWarrior to program in several languages, including C/C++, Pascal, and Java. Third-party compilers provide support for other languages, such as Fortran.

Which languages are available to you depend upon the target for which you are developing software.

Multiple targets You can use CodeWarrior to write software for several different chips or operating systems. CodeWarrior products support programming for a variety of embedded processors and real-time operating systems, the Java Virtual Machine, and desktop operating systems such as Windows and Mac OS.

Most features of CodeWarrior apply regardless of your preferred host, language, or target. General features of CodeWarrior are described in other manuals, such as the *IDE User Guide* and *Debugger User Guide*.

However, each target has its own unique features. This manual describes those unique features.

For a complete understanding of CodeWarrior, you must refer to both the general documentation and the documentation that is specific to your particular target, such as this manual.

The documentation is organized so that various chapters in this manual are extensions of particular generic manuals, as shown in [Table 1.1](#). For a complete discussion of a particular subject, you may need to look in both the generic manual and the corresponding chapter in this Targeting manual.

Table 1.1 CodeWarrior documentation organization

This chapter...	Extends...
CodeWarrior for Win32: A Tutorial	<i>Core Tutorials</i>
Creating Win32/x86 Applications	<i>IDE User Guide</i>
Creating Static Libraries for Win32/x86	
Creating Win32/x86 Dynamic Link Libraries	
Target Settings for Win32/x86	
Debugging for Win32/x86	<i>Debugger User Guide</i>
C and C++ for Win32/x86	<i>C Compilers Reference</i>
Inline x86 Assembler	

This chapter...	Extends...
Pascal for Win32/x86	<i>Pascal Compilers Reference</i>
Libraries and Runtime Code for Win32/x86	<i>MSL C Reference</i> <i>MSL C++ Reference</i>

For example, to completely understand the C/C++ compiler, you need to know the information in the *C Compilers Reference* (which covers the C/C++ front-end compiler) and the information in the [C and C++ for Win32/x86](#) chapter in this manual, which covers the back-end compiler that generates target-specific code.

What's in This Manual

[Table 1.2](#) lists every chapter in this manual, and describes the information contained in each. However, this manual only contains information specific to Win32/x86 software development. See [“CodeWarrior and Its Documentation” on page 11](#) for a discussion of how these chapters relate to other CodeWarrior documentation.

Table 1.2 **Contents of chapters**

Chapter	Description
Introduction	this chapter
Getting Started	installation and setup instructions, and an overview of CodeWarrior
CodeWarrior for Win32: A Tutorial	learn how to use the tools for this target
Creating Win32/x86 Applications	how to build applications for Win32/x86
Creating Static Libraries for Win32/x86	how to build libraries for Win32/x86
Creating Win32/x86 Dynamic Link Libraries	how to build shared libraries for Win32/x86

Introduction

Where to Go from Here

Chapter	Description
Creating Win32 Resources	how to create resources for Win32/x86 projects
Target Settings for Win32/x86	how to control the compiler and linker for this target
C and C++ for Win32/x86	details of the back-end C/C++ compiler for Win32/x86 development
Pascal for Win32/x86	details of the back-end Pascal compiler for Win32/x86 development
Inline x86 Assembler	details support for inline assembly in the compiler
Debugging for Win32/x86	includes details on debugger settings panels for Win32/x86
Libraries and Runtime Code for Win32/x86	libraries provided with CodeWarrior for this target
Troubleshooting	troubleshooting information specific to this target

Where to Go from Here

You'll find all the manuals mentioned in this section on the CodeWarrior CD.

For everyone:

- See the *IDE User Guide* and the *Debugger User Guide* for complete information about the CodeWarrior integrated development environment and debugger.
- For information specific to a particular source language, see the *C Compiler Reference* and the *Pascal Compiler Guide*.
- For information on CodeWarrior's standard libraries for use with a particular source language, see the *MSL C Reference*, the *MSL C++ Reference*, and the *Pascal Language Reference*.

If you're new to CodeWarrior:

- Look for the CodeWarrior tutorials. For the Win32/x86-hosted version of CodeWarrior, these are currently in the form of HTML files.

For beginners: If you're new to Win32 programming, you may find the tutorial chapter of this manual particularly valuable. It provides a step-by-step tutorial for building a simple program to run on Win32 systems.

For reference information on Win32 programming:

- The CodeWarrior Documentation folder on the CodeWarrior Reference CD includes full reference documentation, licensed from Microsoft, on the Win32 Application Programming Interface (API) and the Microsoft Foundation Classes (MFC) application framework.
- *Discover Programming* can teach you how to program in C/C++ or Pascal. You can learn more about Discover Programming at:

<http://www.metrowerks.com/discover/>

Introduction

Where to Go from Here



Getting Started

This chapter gives you the information you need to get CodeWarrior installed and running. For new CodeWarrior users, this chapter provides a brief overview of the CodeWarrior development environment, as well as a description of the development process in CodeWarrior as compared to a command-line environment.

This chapter includes the following topics:

- [System Requirements](#) — hardware and software requirements
- [Installing CodeWarrior for Win32/x86](#) — how to install the various tools
- [CodeWarrior Compiler Architecture](#) — an overview of how front-end and back-end compilers work together
- [Development Tools for Win32/x86](#) — an overview of the CodeWarrior IDE and its components
- [Microsoft Compatibility](#) — how programming with CodeWarrior produces files that are compatible with tools provided by Microsoft
- [The Development Process with CodeWarrior](#) — how programming in CodeWarrior compares to command-line tools

System Requirements

- An IBM-compatible 486 or higher computer. For best performance, a Pentium-class processor is recommended.
- Windows 98, Windows 95, or Windows NT 4.0 operating system
- A minimum of 16 MB RAM. 24MB or 32MB RAM is preferred. When using the CodeWarrior IDE and debugger, 16MB may not provide the best performance.

Getting Started

Installing CodeWarrior for Win32/x86

- A CD-ROM drive to install CodeWarrior software, documentation, and examples.

Installing CodeWarrior for Win32/x86

Your first step toward developing software for your target is to install the CodeWarrior tools. The tools include a variety of components such as the IDE, debugger, plug-in compilers and linkers, standard libraries, runtime libraries and headers, and all necessary documentation.

The CodeWarrior Installer automatically installs all necessary components. It is *strongly* recommended that you use the CodeWarrior Installer to ensure that you have all the required files. If you have any questions regarding the installer, read the instructions built into the CodeWarrior Installer itself.

To start the installation process, double-click the My Computer icon on the Windows desktop, then double-click the drive that holds the CodeWarrior CD. When you do, a dialog box should appear with a button titled “Launch CodeWarrior Setup.” Click this button. If for some reason you don’t see this dialog, double-click the `setup.exe` file located at the root level of the CD.

CodeWarrior Compiler Architecture

A proprietary multi-language, multi-target compiler architecture is at the heart of CodeWarrior. Front-end language compilers generate a memory-resident, unambiguous, language-independent intermediate representation (IR) of syntactically correct source code. Back-end compilers generate code from the IR for specific targets. The CodeWarrior IDE manages the whole process.

As a result of this architecture, the same front-end compiler is used in support of multiple back-end targets. In some cases, the same back-end compiler can generate code from a variety of languages.

All compilers are built as plug-in modules. The interface between the IDE and compilers and linkers is public, so third parties can create compilers that work with CodeWarrior.

Once the compiler generates object code, the plug-in linker generates the final executable. Some targets have multiple linkers available to support different object code formats.

Development Tools for Win32/x86

Programming for Win32 is much like programming for any other target in CodeWarrior. If you have never used CodeWarrior before, the tools you will need to become familiar with are:

- [CodeWarrior IDE](#)
- [CodeWarrior Debugger for Win32/x86](#)
- [CodeWarrior C/C++ Compiler for Win32/x86](#)
- [CodeWarrior Pascal Compiler for Win32/x86](#)
- [Resource Compiler for Win32](#)
- [CodeWarrior Linker for Win32/x86](#)
- [Metrowerks Standard Libraries](#)
- [Win32 API Headers](#)
- [MFC Application Framework](#)

If you are an experienced CodeWarrior user, this is the same IDE and debugger that you've been using all along. You will, however, need to become familiar with the Win32/x86 runtime software environment.

CodeWarrior IDE

The CodeWarrior IDE is the application that allows you to write your software. It controls the project manager, the source code editor, the class browser, the compilers and linkers, and the debugger.

The project manager may be new to those more familiar with command-line development tools. All files and settings related to your

Getting Started

Development Tools for Win32/x86

project are organized in the project manager. This allows you to see your project at a glance, and eases the organization of and navigation among your source code files. The IDE manages all build dependencies.

A project may also have multiple “targets.” In this context, a target is a separate build (with its own settings) that uses some or all of the files in the project. For example, you can have a debug version and a release version of your software as separate targets in the same project.

For more information about how the CodeWarrior IDE compares to a command-line environment, see [“The Development Process with CodeWarrior” on page 25](#). That short section discusses how various parts of the IDE implement the classic features of a command-line development system based on makefiles.

The CodeWarrior IDE has an extensible architecture that uses plug-in compilers and linkers to target various operating systems and microprocessors. The CodeWarrior CD includes a C/C++ compiler for the Win32/x86 family of processors. Other CodeWarrior packages include C, C++, Pascal, and Java compilers for Win32, Mac OS, and other platforms.

For more information about the CodeWarrior IDE, you should read the *IDE User Guide*.

CodeWarrior Debugger for Win32/x86

The CodeWarrior debugger controls your program’s execution and allows you to see what’s happening internally as your program runs. You use the debugger to find problems in your program’s execution.

The debugger can execute your program one statement at a time, and suspend execution when control reaches a specified point. When the debugger stops a program, you can view the chain of function calls, examine and change the values of variables, and inspect the contents of the processor’s registers.

For general information about the debugger, including all of its common features and its visual interface, you should read the *Debugger User Guide*.

For more information about debugging software for Win32/x86, see [“Debugging for Win32/x86” on page 127](#).

The Mac-OS–hosted version of the CodeWarrior debugger includes remote debugging capabilities for Win32/x86 applications. This is a two-machine debugger that runs on the Mac OS host and communicates with the running application via a remote communications nub on the x86 target computer, using either a serial connection or a TCP/IP network connection. Using the remote debugger requires some setup. You’ll find full information in [“Setting Up for Remote Debugging” on page 130](#).

CodeWarrior C/C++ Compiler for Win32/x86

The CodeWarrior C/C++ compiler for Win32/x86 is an ANSI compliant C/C++ compiler. This compiler is based on the same compiler architecture that is used in all of the CodeWarrior C/C++ compilers. When used with the CodeWarrior linker for Win32/x86, you can generate Win32/x86 applications and libraries that conform to Microsoft’s specifications.

For more information about the CodeWarrior C/C++ language implementation, you should read the *C Compilers Reference*. For information on the Win32/x86 back-end compiler, see [“Target Settings for Win32/x86” on page 79](#) and [“C and C++ for Win32/x86” on page 101](#).

In addition, the C/C++ compiler supports inline assembly for Win32/x86 development. See [“Inline x86 Assembler” on page 123](#).

CodeWarrior Pascal Compiler for Win32/x86

The CodeWarrior Pascal compiler for Win32/x86 is an ANS compliant Pascal compiler. This compiler is based on the same compiler architecture that is used in all of the CodeWarrior compilers. When used with the CodeWarrior linker for Win32/x86, you can generate

Getting Started

Development Tools for Win32/x86

Win32/x86 applications and libraries that conform to Microsoft's specifications.

For more information about the CodeWarrior Pascal language implementation, you should read the *Pascal Compilers Reference*. For information on the Win32/x86 back-end compiler, see [“Target Settings for Win32/x86” on page 79](#) and [“Pascal for Win32/x86” on page 111](#).

In addition, the Pascal compiler supports inline assembly for Win32/x86 development. See [“Inline x86 Assembler” on page 123](#).

Resource Compiler for Win32

The Win32 resource compiler (RC) is another plug-in tool to the CodeWarrior IDE. It reads standard Win32 resource scripts and compiles them into resources that are then linked into your project's object file. See [“Win32 Resource Compiler” on page 76](#) for more information.

CodeWarrior Linker for Win32/x86

Like the compilers, the CodeWarrior Win32/x86 linker is a plug-in tool integrated into the CodeWarrior IDE. It links multiple object modules and libraries, resolving references from one to another to produce executable applications, dynamic link libraries (DLLs), or static libraries.

When you execute a **Make** command to build your project and any of the project's object files or libraries have changed since the last build, CodeWarrior automatically invokes the linker as the last step in the build process.

When linking your project, the linker reads these files:

- COFF (Common Object File Format) object modules produced by the x86 compiler or the Win32 resource compiler. These modules are stored in the project file and are not explicitly visible.

- COFF object modules that you add explicitly to the project. Any file you add that has the extension `.obj` is treated as a COFF object.
- COFF library files you add to the project. Any file with the extension `.lib` is treated as a COFF library.
- A linker command file, which is a text file containing commands to be processed by the linker.

The linker can generate any of the following as output:

- A COFF/PE executable file (file extension `.exe`). This is the standard-format executable file for all Win32 systems.
- A COFF/PE dynamic link library, or DLL (file extension `.dll`). DLLs are in the same format as applications, but do not run on their own. They are used by other applications to link dynamically to other code at load or run time.
- A COFF Library (LIB) file.

For more information about the linker, see [“x86 Linker” on page 92](#).

Metrowerks Standard Libraries

The Metrowerks Standard Libraries (MSL) are standard C and C++ libraries for use in developing applications for Win32/x86. The libraries are ANSI compliant, and all of the source for the libraries is provided for you to use in your projects. These are the same libraries that are used for all CodeWarrior targets, but they have been customized and the runtime has been adapted for use in Win32/x86 development.

For more information about MSL, you should read the *MSL C Reference* and the *MSL C++ Reference*. To learn how MSL has been adapted for use in Win32/x86 applications, you should read [“The Metrowerks Standard Libraries” on page 135](#).

Win32 API Headers

The Win32 Application Programming Interface (API) is a set of data structures and functions used to interface to 32-bit versions of the Windows operating system. The Win32 API was designed for use

with C, Pascal, and other languages. CodeWarrior provides a complete set of header files and libraries that allow you to use all Win32 API functions from either C/C++ or Pascal.

MFC Application Framework

The Microsoft Foundation Classes (MFC) are an object-oriented application framework that implements the key building blocks in a Win32 application. CodeWarrior includes the full MFC source.

Microsoft Compatibility

CodeWarrior's Win32/x86 development tools strive for maximum compatibility with the tools provided by Microsoft:

- CodeWarrior's object modules, object libraries, executables, and dynamic link libraries use the same file formats as those produced by Microsoft (for instance, COFF objects, COFF/PE executables, and DLLs).
- CodeWarrior compilers duplicate the function-calling and name-mangling conventions used by Microsoft compilers.
- CodeWarrior's C data structures are laid out the same way as Microsoft's.
- CodeWarrior's resource files are in the same format as Microsoft's.
- C++ classes that use only single inheritance are compatible between CodeWarrior and Microsoft's Visual C++.
- The CodeWarrior Win32/x86 compiler and linker support Microsoft CodeView debugging information.
- Optional C++ exception handling is compatible with the scheme used by Microsoft Visual C++.

All this compatibility means you can do any of the following:

- Link Microsoft-created object modules and libraries with the CodeWarrior linker.
- Link CodeWarrior-created object modules with the Microsoft linker.

- Use CodeWarrior to develop code for Microsoft's OLE (Object Linking and Embedding) programming model.
- Use dynamic link libraries created by either CodeWarrior or Microsoft's tools interchangeably.
- Use Microsoft Visual C++ Developer's Studio to debug your CodeWarrior-generated code. You can also use third-party debuggers and tools such as Nu-Mega Technologies' SoftICE and BoundsChecker.

For now, we have to admit to a few incompatibilities as well:

- C++ multiple-inheritance classes are not compatible with Microsoft's, so you can't mix such classes from the two compilers.
- CodeWarrior does not support the older object format used by Borland and Microsoft prior to Visual C++. Such objects must be converted to COFF format before CodeWarrior will accept them.
- CodeWarrior does not yet support Microsoft-compatible Run Time Type Information (RTTI).

The Development Process with CodeWarrior

While working with CodeWarrior, you will proceed through the development stages familiar to all programmers: write code, compile and link, debug. For complete information on performing tasks like editing, compiling, and linking, refer to the *IDE User Guide*. For debugging, see the *Debugger Guide*.

The difference between CodeWarrior and traditional command line environments is in how the software (in this case the IDE) helps you manage your work more effectively. If you are unfamiliar with an integrated environment in general, or with CodeWarrior in particular, you may find the topics in this section helpful. Each topic discusses how one component of the CodeWarrior tools relates to a traditional command-line environment.

Read these topics to find out how using the CodeWarrior IDE differs from command-line programming.

Getting Started

The Development Process with CodeWarrior

- [Makefiles](#)—the IDE uses a project to control source file dependencies and the settings for compilers and linkers
- [Editing Code](#)—an overview of source code editing in the IDE
- [Compiling](#)—how the IDE performs compile operations
- [Linking](#)—how the IDE performs linking operations
- [Debugging](#)—how to debug a program from the IDE
- [Viewing Preprocessor Output](#)—A tip on debugging preprocessor directives

Makefiles

The CodeWarrior IDE *project* is analogous to a makefile. Because you can have multiple builds in the same project, in fact the project is analogous to a collection of makefiles. For example, you can have one project that has both a debug version and a release version of your code. You can build one or the other, or both as you wish. In CodeWarrior, these different builds within a single project are called “targets.”

The IDE uses the project manager window to list all the files in the project. Among the kinds of files in a project are source code files and libraries.

You can add or remove files easily. You can assign files to one or more different targets within the project, so files common to multiple targets can be managed simply.

The IDE manages all the interdependencies between files automatically, and tracks which files have been changed since the last build. When you rebuild, only those files that have changed are recompiled.

The IDE also stores the settings for compiler and linker options for each target. You can modify these settings using the IDE, or with `#pragma` statements in your code.

Editing Code

The CodeWarrior IDE has an integral text editor designed for programmers. It handles text files in MS-DOS/Windows, UNIX, and Mac OS formats.

To edit a source code file, or any other editable file that is in a project, just double-click the file's name in the project window to open the file.

The editor window has excellent navigational features that allow you to switch between related files, locate any particular function, mark any location within a file, or go to a specific line of code.

Compiling

To compile a source code file, it must be among the files that are part of the current target. If it is, you simply select it in the project window and choose **Compile** from the Project menu.

To compile all the files in the current target that have been modified since they were last compiled, choose **Bring Up To Date** in the Project menu.

In UNIX and other command-line environments, object code compiled from a source code file is stored in a binary file (a ".o" or ".obj" file). The CodeWarrior IDE stores and manages object files transparently.

Linking

Linking object code into a final binary file is easy: use the **Make** command in the Project menu. The **Make** command brings the active project up to date, then links the resulting object code into a final output file.

You control the linker through the IDE. There is no need to specify a list of object files. The project manager tracks all the object files automatically. You can use the project manager to specify link order as well.

Getting Started

The Development Process with CodeWarrior

Use the [x86 Target](#) settings panel to set the name of the final output file.

Debugging

To debug a project, make sure **Enable Debugging** is selected in the Project menu.

For more information on debugging, refer to the *Debugger User Guide*. For information specific to Win32/x86 debugging, see [“Debugging for Win32/x86” on page 127](#).

Viewing Preprocessor Output

To view preprocessor output, select the file in the project window and choose **Preprocess** from the Project menu. A new window appears that shows you what your file looks like after going through the preprocessor.

You can use this feature to track down bugs caused by macro expansion or other subtleties of the preprocessor.



CodeWarrior for Win32: A Tutorial

This tutorial is designed to give you a quick start at learning how to use CodeWarrior for Win32/x86 programming. These are the topics covered in this chapter:

- [Phase 1: Creating a Project](#)
- [A Look at the Code](#)
- [Phase 2: Getting Resourceful](#)
- [Phase 3: Handling Commands](#)
- [Phase 4: What, Pray Tell, Is a DLL?](#)
- [Phase 5: All Together Now](#)

Tutorial: Meet Ray the Tracer

In this tutorial, you'll create a ray-tracing application program. Ray tracing is a method of rendering three-dimensional scenes with photo-realistic accuracy, including shading, shadows, and reflection of light. It operates by tracing the path from your eye to objects, and from the objects to the sources of light. The process is highly processor-intensive, requiring a significant amount of computation for each pixel displayed.

TIP: For best results when running your application, have your monitor set to display at least 16 thousand colors.

To give you some useful experience, the user interface for the program will be a Microsoft Foundation Classes (MFC) application, and the ray-tracing engine will be a dynamic link library (DLL). For more information, see

- [“The Microsoft Foundation Classes” on page 139](#)
- [“About Dynamic Link Libraries” on page 65](#)

See also *TheIDE User Guide*

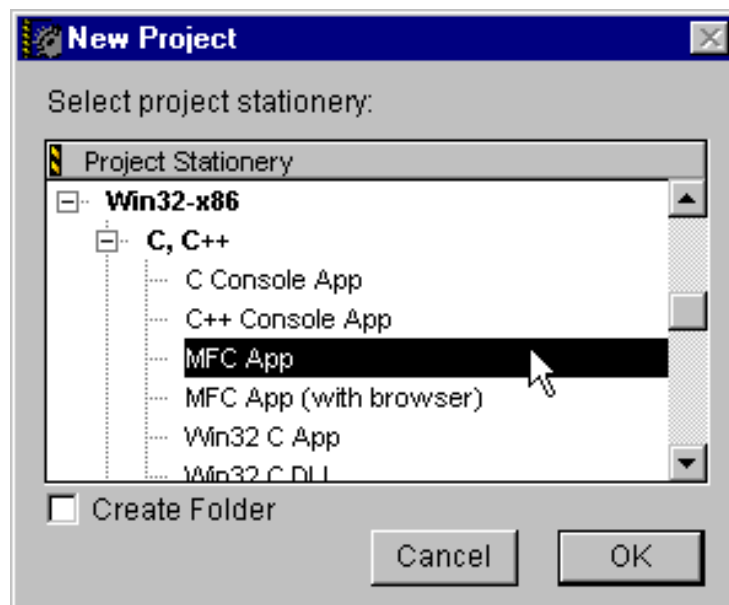
Phase 1: Creating a Project

Before you begin, copy the RayTracer folder from the CodeWarrior CD to your hard disk. You'll find it in the CodeWarrior Examples folder, in the Win32 Examples subfolder under Win32 Tutorial. Open the copy of the folder on your hard disk, start up the CodeWarrior IDE, then follow these steps to set up the project:

1. Choose the **New Project** command from the **File** menu.

The **New Project** dialog box ([Figure 3.1](#)) will appear.

Figure 3.1 New Project dialog with Win32-x86 stationery



2. Disable the Create Folder checkbox.

You need to select the Create Folder checkbox only when you want to create a folder in which to put the project you are creating. A few steps later, you will place the project in an existing folder.

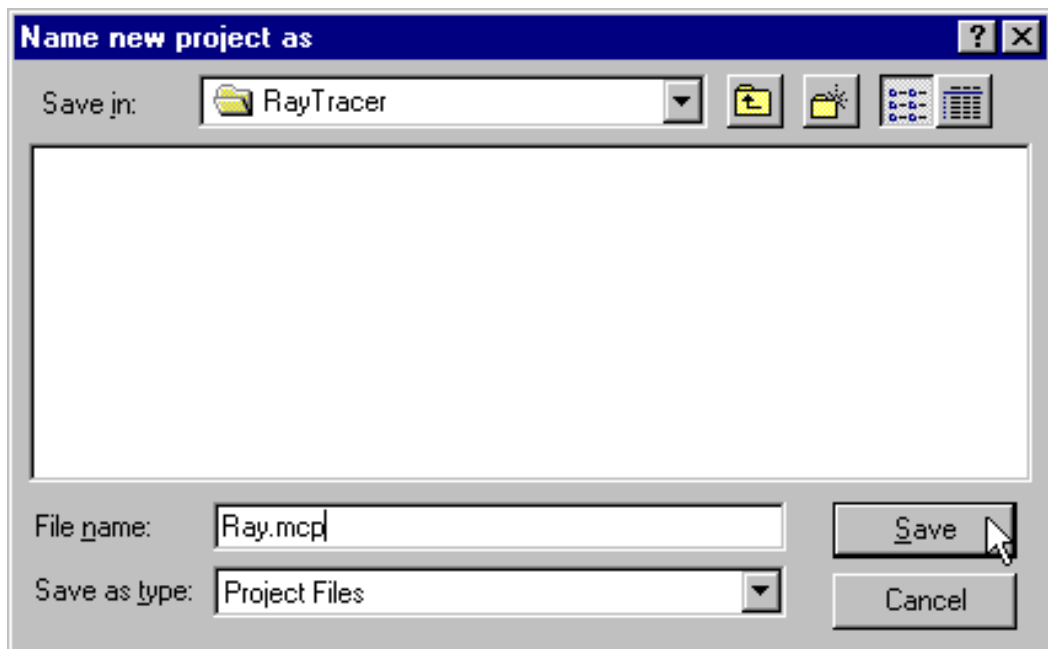
3. Display the list of Win32-x86 project stationery.

Click the tree control icon next to the line **Win32-x86** until it looks like [Figure 3.1](#). The list will expand to show the Win32-x86 stationery files available.

4. Select your stationery.

Select the **MFC App** stationery and click **OK**. You'll see the **Name new project** dialog box ([Figure 3.2](#)).

Figure 3.2 Name new project dialog



5. Locate the destination folder.

Navigate to the copy of the RayTracer folder that you made on your hard disk.

CodeWarrior for Win32: A Tutorial

Phase 1: Creating a Project

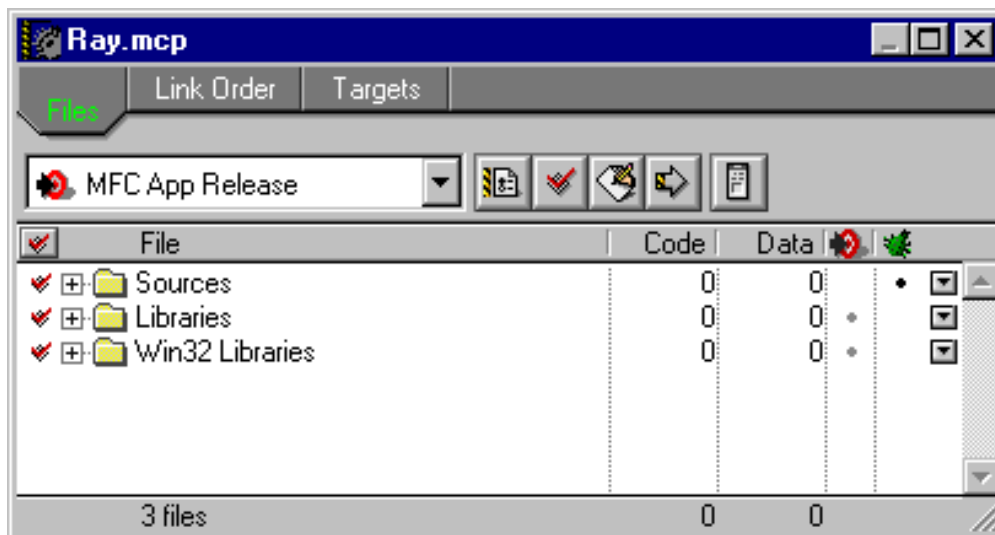
6. **Name your project.**

Type the file name `Ray.mcp` into the text box labeled **File name**.

7. **Save the project.**

Click the **Save** button on the dialog box. The project window ([Figure 3.3](#)) will appear, listing the files the **MFC App** stationery automatically included in the project for you.

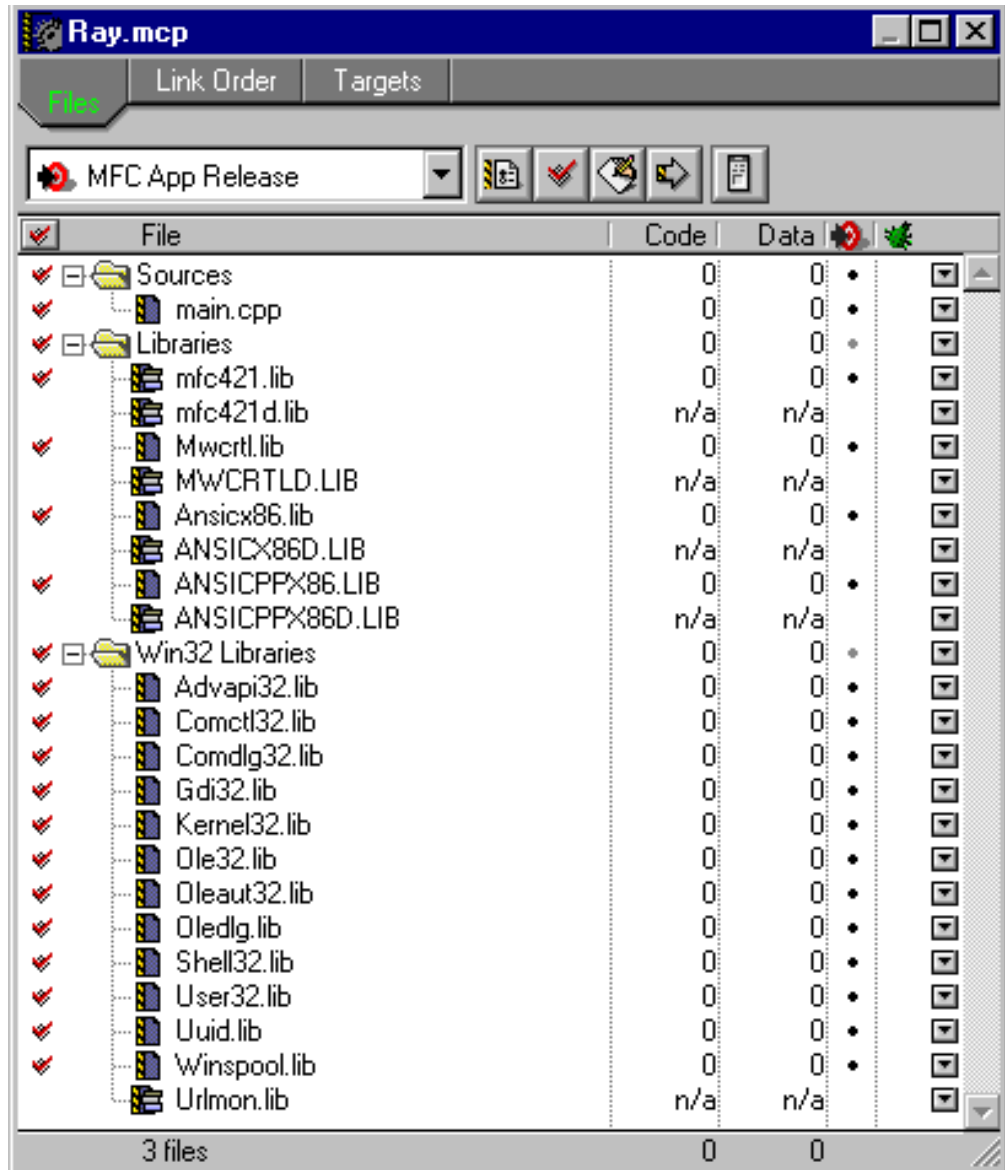
Figure 3.3 Ray.mcp project window



8. **Open the folders to see all files in the project.**

Click in the box with “+” next to each folder to view files, as shown in [3.4](#).

Figure 3.4 Ray.mcp project window displaying all files



9. Add the program's source file to your project.

Click on the `main.cpp` file to select it. Choose **Add Files** from the **Project** menu. Navigate to the RayTracer folder, select `Ray1.cpp`,

CodeWarrior for Win32: A Tutorial

Phase 1: Creating a Project

and click **Add** ([Figure 3.5](#)). Add the file to all targets and click **OK** ([Figure 3.6](#)). This places `Ray1.cpp` in the same spot as `main.cpp`.

Figure 3.5 Select files to add dialog

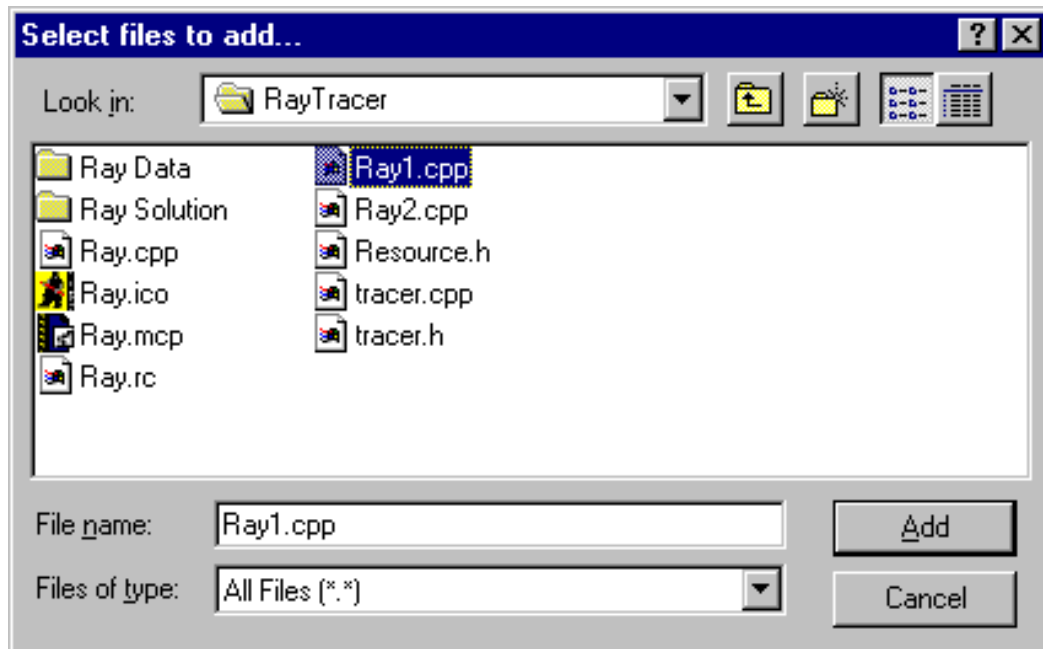
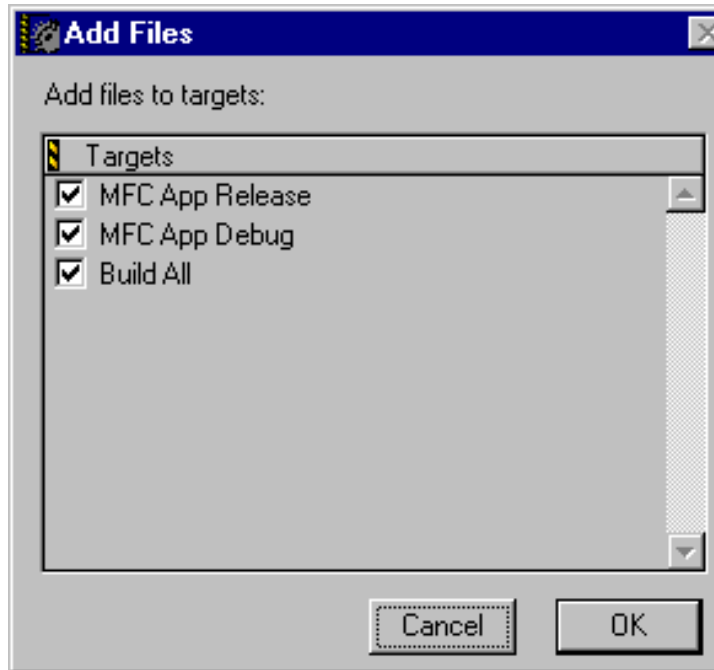


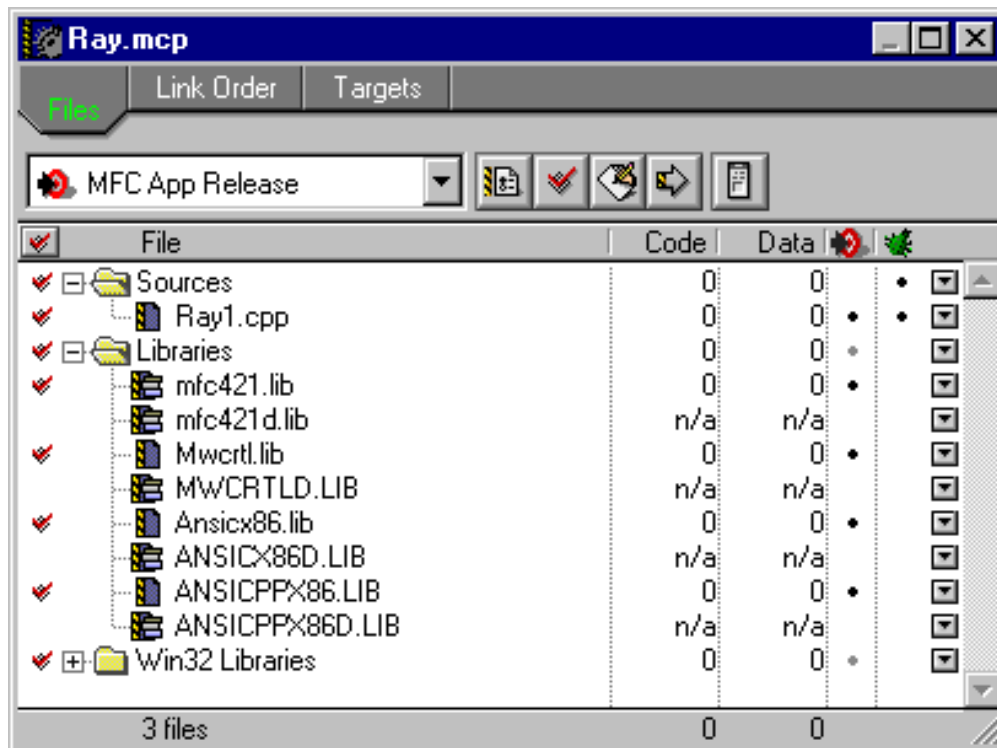
Figure 3.6 Add Files dialog



10. Remove the placeholder from your project window.

Select the file `main.cp` and choose **Remove Selected Items** from the Project menu. Your project window should now look like [Figure 3.7](#).

Figure 3.7 Project window with source file added

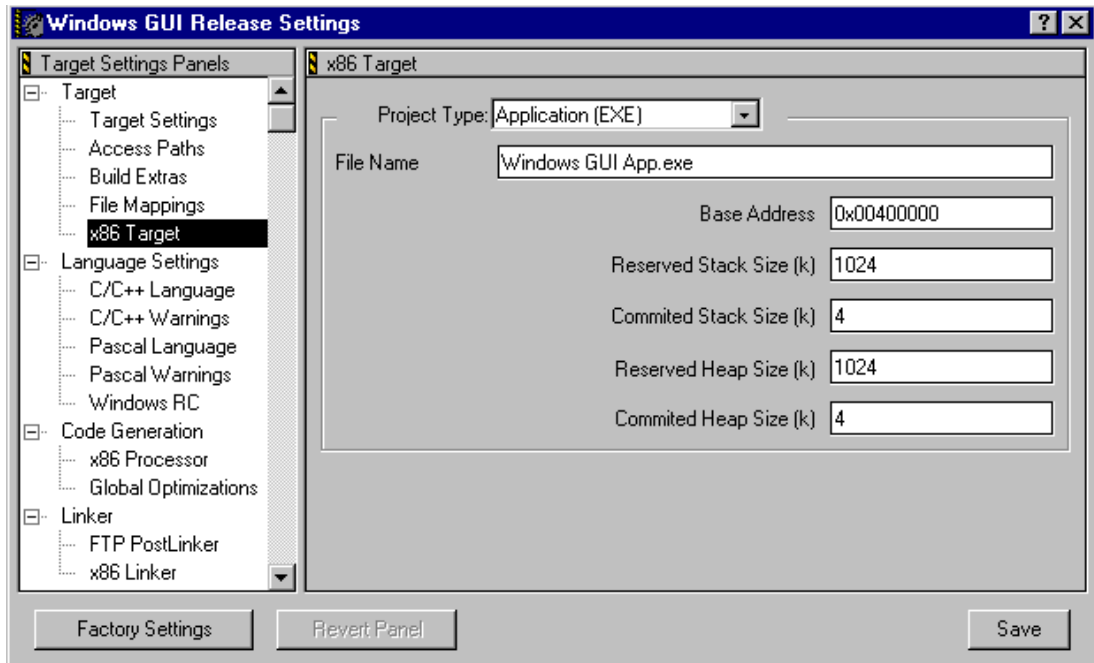


11. Display the Target Settings dialog box.

Choose *MFC App Settings* command from the Edit menu. CodeWarrior will display the **Target Settings** dialog box, listing a variety of setting panels for specifying attributes of your project.

12. Select the x86 Target panel.

Click on **x86 Target** in the list of setting panels to the left. The **x86 Target** setting panel appears, as shown in [Figure 3.8](#).

Figure 3.8 Target Settings dialog box showing x86 Target panel

13. Specify the name and type of your output file.

Make sure **Application (EXE)** is selected from the pop-up menu labeled **Output File Type**, and type the name **Ray.exe** into the **File Name** text box. Then click the **Save** button to confirm your settings and close the dialog.

14. Build the program.

Choose **Make** from the Project menu. CodeWarrior will proceed to compile and link your program.

15. Run the program.

Choose **Run** from the Project menu. If all has gone well, it should open an empty window titled **Ray the Tracer** and let you manipulate the window on your screen: move it, resize it, maximize it, minimize it, and close it.

A Look at the Code

You've just built the first version of Ray the Tracer, which at this point just puts up an empty window on the screen. Let's take a look at the code.

In your Ray.mcp project window, double-click the file name Ray1.cpp to open a window displaying the contents of the file. At the top of the source file, you'll see the line

```
#include <afxwin.h>
```

The header file `afxwin.h` defines the most common of the Microsoft Foundation Classes (MFC). MFC is an object-oriented interface to Win32, consisting of classes you either use directly in your programs or from which you derive subclasses of your own.

The next line of code,

```
class CTracerWindow: public CFrameWnd
```

defines a class for our main window, `CTracerWindow`. This class is derived from the MFC class `CFrameWnd`, which implements the behavior of a standard Win32 window. The only member function in our derived class is the constructor, `CTracerWindow`. Looking down a little further at the definition of this function, you can see that it simply calls the `Create` function, inherited from the base class `CFrameWnd`, to create the window.

The line

```
class CTracerApp: public CWinApp
```

defines a class derived from `CWinApp`, the MFC class representing a Win32 application. Its only member function is `InitInstance`, called during application startup. Our definition of this function creates a new window of class `CTracerWindow` and assigns it to member variable `m_pMainWnd`, inherited from the base class `CWinApp`. The next two statements display the window and update (paint) its contents on the screen. Finally, `InitInstance` returns `TRUE` to indicate that it has succeeded.

This last line in the source file,

```
CTracerApp TracerApp;
```

may look innocuous, but it actually is responsible for the entire execution of the program. Here's what MFC does behind the scenes:

1. It declares a variable of type `CTracerApp`.
2. The constructor for this object calls the `InitInstance` function we just looked at.
3. `InitInstance`, in turn, creates the window.
4. The `CTracerApp` constructor then automatically initiates a message loop, which runs for the duration of the application.
5. Finally, the message loop receives input events (clicks and keystrokes) from the mouse and keyboard and dispatches it to the code that handles these events.

Currently, our program doesn't contain code to handle events, so it can't do very much. However, MFC provides defaults that allow the user to manipulate the window on the screen in all the usual ways: move it, resize it, maximize it, minimize it, or close it. That's actually quite a lot for such a simple application. The main thing is, it's a working program. If getting started is the hardest part of a project, that part, at least, is done.

Now that we're finished examining the code, you can go ahead and close the `Ray1.cpp` window on your screen.

Phase 2: Getting Resourceful

Most Win32 programs use *resources* to define things like menus, icons, and other objects the program uses. In this phase, we'll add a resource file to our application project.

1. **Open the `Ray.mcp` project window.**

If the project window isn't already open on your screen, you can use the **Open** command on CodeWarrior's File menu to open it, or double-click its icon in Windows Explorer.

2. **Open the source file.**

Double-click the file name `Ray1.cpp` in the project window to display the contents of the file.

3. Include the header file defining the resource identifiers.

After the line

```
#include <afxwin.h>
```

at the beginning of the source code, add the line

```
#include "resource.h"
```

4. Modify the code to create its window from a resource.

In the definition of the constructor function `CTracerWindow`, replace the statement

```
Create(NULL, _T("Ray the Tracer"),  
        WS_OVERLAPPEDWINDOW, rectDefault);
```

with

```
// Create the window with attributes from ray.rc  
LoadFrame(IDR_MAINFRAME);
```

In Win32 resource files, each resource has an integer or string identifier. The `LoadFrame` function (inherited from base class `CWndFrame`) looks for resources with a specified identifier, then creates a window based on the information in those resources. In this case, it will find the window's menu, a string for the window's title, and an icon, all of which have the identifier `IDR_MAINFRAME`.

But where do these resources reside? In a resource file, of course, in this case named `Ray.rc`. To make them available, we must add that file to the project.

5. Add the resource file to the project.

Drag the file `Ray.rc` from the `Ray the Tracer` folder and drop it in the **Sources** group of the project window, or use the **Add Files** command on CodeWarrior's Project menu to add it to the project.

Let's take a moment to look at the contents of the resource file. Open the resource file by double-clicking its name, `Ray.rc`, in the project window. You'll see that it begins with the line

```
#include "resource.h"
```

which includes the same header file as in `Ray1.cpp`. Next is a line reading

IDR_MAINFRAME MENU DISCARDABLE

This introduces the definition of the program's main menu. The lines that follow specify the menu's contents. In this case, since this is a main menu, its items are the names of the individual menus in the window's menu bar.

- The lines labeled **POPUP**, which are followed by a **BEGIN** and **END**, define the menus attached to each of the menu bar items.
- The **MENUITEM** lines between **BEGIN** and **END** specify the items on each popup menu.
- The name at the end of each **MENUITEM** line is the identifier for the command that will be sent to the application when that menu item is selected.
- Finally, the **STRINGTABLE** definition specifies strings that the program uses. The one labeled **IDR_MAINFRAME** is used automatically as the title of the window when the program calls **LoadFrame**.

Now we're ready to build the new version of the program.

6. **Build the program.**

As before, choose **Make** from the Project menu to compile and link the modified version of the program.

7. **Run the new program.**

Run it to verify that it displays the window's menu bar. You may notice that all commands but the **Exit** command on the File menu are grayed out. That is because the program doesn't yet contain any code to handle these commands; MFC is clever enough to recognize this and disable the commands that the program has left unimplemented.

Phase 3: Handling Commands

Now we're ready to make the program a bit more interesting by handling the commands on its menus.

1. **Open the project window.**

Once again, if the project window isn't already open on your screen.

2. **Replace the source file `Ray1.cpp` with `Ray2.cpp`.**

Choose **Add Files** from the Project menu to add `Ray2.cpp` to the project. Then select `Ray1.cpp` in the project window and remove it with the Project menu's **Remove Selected Items** command.

3. **Display the contents of the new source file.**

Double-click `Ray2.cpp` in the project window to open the file on your screen.

Looking at the code, you can see that we've added several new member functions to class `CTracerWindow`, such as `OnPaint()` and `ViewFront()`. `OnPaint()`, an MFC member function inherited from the base class `CFrameWnd`, will be called whenever our window receives a paint message, indicating that the window's contents need to be repainted on the screen. The other new functions, such as `ViewFront()`, handle menu commands that the user chooses from the window's View menu.

Following the function declarations is the line

```
DECLARE_MESSAGE_MAP()
```

This is an MFC macro that creates the window's *message map*. The message map tells MFC how to map incoming messages sent to the window into calls to the appropriate member functions. The actual definition of the message map is accomplished by the following series of macro calls below the `CTracerWindow` class declaration:

```
BEGIN_MESSAGE_MAP(CTracerWindow, CFrameWnd)
    ON_WM_PAINT()
    ON_COMMAND(ID_VIEW_FRONT), ViewFront()
    ON_COMMAND(ID_VIEW_LEFT), ViewLeft()
    ON_COMMAND(ID_VIEW_RIGHT), ViewRight()
    ON_COMMAND(ID_VIEW_BACK), ViewBack()
END_MESSAGE_MAP()
```

The `ON_WM_PAINT` macro specifies that our member function `OnPaint()` is to be called whenever the window gets a paint message. The next four lines identify the member functions that handle each of the four commands on the View menu. The identifiers defined in

the resource header file `Ray.rc`, such as `ID_VIEW_FRONT`, are used to connect each menu command with the corresponding member function via the `ON_COMMAND` macro.

Right after the message map are the lines

```
CDC Picture;  
CBitmap Bitmap;
```

These are declarations for two MFC objects: a display context and a bitmap. Together, they allow us to draw a picture and display it in our window. How this actually works is beyond the scope of this tutorial, but we can still have a peek at the code.

Down a little further is the function definition for the member function `OnPaint()` of class `CTracerWindow`. This is the handler function for paint messages, the essential part of the program that actually puts something on the screen. This function calls `GetClientRect()` to get the dimensions of the window's client area (the area the program can draw into), then copies the bitmap (which will eventually contain our ray-traced image) into the window, using the function `BitBlt()`.

NOTE: The functions `GetClientRect()` and `BitBlt()` belong to the Win32 application program interface (API). We're using them here as MFC member functions. MFC member functions generally map directly into the corresponding API calls. If we were not using MFC, we would call the API functions directly.

The remaining member functions we've added to `CTracerWindow` (`ViewRight()`, `ViewLeft()`, `ViewFront()`, and `ViewBack()`) will eventually implement the actual ray-tracing procedures. For now, they're simply left empty.

4. **Build the program.**

Use the **Make** command on the Project menu to compile and link this latest version of the Ray the Tracer program.

5. **Run the program.**

When you run the program, you'll see a black box, 200 pixels wide by 200 high, appear in the middle of your window. This is the (cur-

rently empty) bitmap that the `BitBlt()` function in our `OnPaint()` function copied into the window. Notice also that the **From Front**, **From Left**, **From Right**, and **From Back** commands on the program's View menu have been enabled, since we've now defined functions to handle them (even though all of these functions are currently empty).

Phase 4: What, Pray Tell, Is a DLL?

We've now completed the (admittedly meager) user interface of our ray-tracing application. All that's missing is some code to do the actual ray tracing. Unless you have a Ph.D. in math (or can at least remember Algebra 2), you'd better not try to write this code yourself. Fortunately, you have a friend (coincidentally named Ray) who offers you a ready-made, generic ray-tracing engine already written in C++.

"The beauty of it," says Ray, "is that this engine is totally system-independent. The truth is, it knows nothing about the hardware or operating system it's running on."

Hmm. We get suspicious when programmers talk about beauty and truth. But in this case, Ray does not exaggerate. The tracer engine defines a class `RayTracer` with a single abstract virtual member function named `SetPixel()`. All you have to do to use the engine is derive a class from `RayTracer` and define its `SetPixel()` function. Sounds easy enough.

It would be a shame to besmirch such beautiful code with the likes of Win32 and MFC, so you decide to keep the tracer engine as independent as possible of your user-interface code. Besides, when Windows 2005 comes along, you might want to transport this code, so there are benefits to keeping it clean.

The solution is to place the code in a *dynamic link library* (DLL). A DLL contains code that does not stand on its own as an application, but instead is used by other applications. It can be kept completely separate, residing in a different file from the applications that use it.

Creating a DLL is very similar to creating an application, so we won't elaborate much on the individual steps except where they differ from the ones we looked at earlier. Here's how to create a DLL from the ray-tracing engine:

1. **Choose the New Project command from the File menu.**

You'll be presented with the **New Project** dialog (shown earlier in [Figure 3.1 on page 30](#)).

2. **Disable the Create Folder checkbox.**

3. **Select the Win32 C++ DLL stationery.**

Click the **Win32-x86** expansion icon to see the list of Win32-x86 project stationery. Select **Win 32 C++ DLL** and click **OK**. You'll see the **Name new project** dialog, as in [Figure 3.2 on page 31](#).

4. **Name the project.**

Navigate to the Ray the Tracer folder and type in the name `Tracer.mcp`. Then click **Save** to dismiss the dialog box and create the project. A new project window will appear, listing the files automatically included in the project by the **Win32 C++ DLL** stationery.

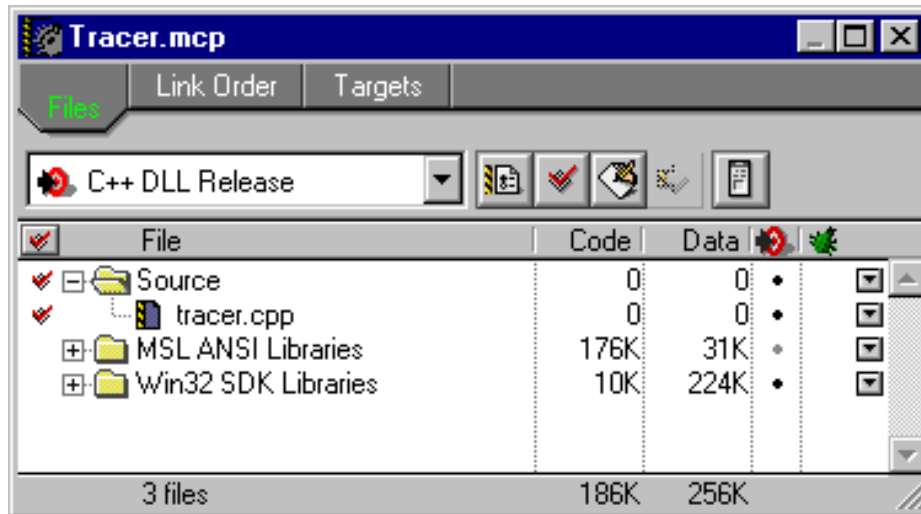
5. **Add the source file `tracer.cpp` to the project.**

Use the **Add Files** command on the Project menu. Then select the placeholder line reading

`x86/Win32 DLL.c`

and remove it from the project with the **Remove Selected Items** command on the Project menu. Your project window should now look like [Figure 3.9](#).

Figure 3.9 Tracer.mcp project window



6. Display the x86 Target setting panel.

Choose **C++ DLL Settings** from the Edit menu to display the **Target Settings** dialog box, then select **x86 Target** from the list of available setting panels.

TIP: You can also display the **Target Settings** dialog box by clicking the Target Settings button, which is the left-most icon button on the project window.

7. Specify the name of the output file.

The file type **Dynamic Link Library (DLL)** is automatically set for you by the **Win32 C++ DLL** stationery. Type the name **Tracer.dll** into the **File Name** text box and click **Save** to confirm your settings.

8. Build the DLL.

Choose **Make** from the Project menu to compile and link the DLL. This will create two files:

- **Tracer.dll**, the actual DLL, containing all the code and resources included in the project.

- **Tracer.lib**, a static library for linking client programs to the DLL. Adding this file to a project makes the contents of the Tracer DLL available to another program (an application or even another DLL).

The DLL communicates with its client programs by *exporting* some of its functions and variables with the declaration modifier `__declspec(dllexport)`. This modifier specifies that the declaration that follows is to be made available to the DLL's client programs. To see an example, double-click the source file `tracer.cpp` in the `Tracer.mcp` project window, to open the file on your screen. Then use the **Find** command on the Search menu to search for the string `__declspec`. (Note the double underscore at the beginning of the name.) You'll find the lines

```
#ifdef __INTEL__
__declspec(dllexport)
#endif
void RayTracer::Trace(Object *ObjectList,
    LightSource *LightList, int w, int h,
    Vector ViewPos)
```

This says that if the code is being compiled for an Intel processor, the function `Trace` is to be exported as part of the DLL's client interface. Use the **Find Next** command from the Search menu and you'll find several more exported names as well.

Phase 5: All Together Now

Now that we actually have some ray-tracing code (in a DLL) and a user interface (in an application), the only thing left to do is link them together. A header file named `Tracer.h` came with your friend Ray's tracing engine. You can add this file to your `Ray.mcp` project to define the interface from your application to the ray tracer. After that, it's only a matter of adding a little "glue" code to hold it all together.

1. **Open the `Ray.mcp` project in the CodeWarrior IDE.**

Use the **Open** command on the File menu.

CodeWarrior for Win32: A Tutorial

Phase 5: All Together Now

2. Add the tracer's interface library to the project.

Use the **Add Files** command on the Project menu.

3. Replace the source file `Ray2.cpp` with `Ray.cpp`.

Again, use the **Add Files** command to add file `Ray.cpp` to the project. Then use **Remove Selected Items** on the Project menu to remove the old source file `Ray2.cpp`.

Display the code by double-clicking file `Ray.cpp` in the project window. The first new section you'll see is shown in [Listing 3.1](#).

Listing 3.1 Object definitions for the ray-tracing program

```
// Some serious balls
Sphere Object1(0.0, 0.0, 0.0, 5.0, 1.0, 0.5, 0.1, 0.5);
Sphere Object2(4.0, 2.0, -4.0, 2.0, 0.3, 1.0, 1.0, 0.5);
Sphere Object3(-4.0, 2.0, -4.0, 2.0, 0.3, 1.0, 1.0, 0.5);

// And a floor
BoundedPlane Object4(0.0, 1.0, 0.0, 6.0, -100.0, 100.0,
    -10000, 10000, -1000.0, 1000.0, 0.5, 0.5, 0.7, 0.2);

// And a ceiling
BoundedPlane Object5(0.0, 1.0, 0.0, -12.0, -100.0, 100.0,
    -10000, 10000, -1000.0, 1000.0, 0.7, 0.5, 0.5, 0.0);

// And a couple of lights
LightSource Light1(10.0, 10.0, -10.0, 50000.0, 50000.0, 50000.0);
LightSource Light2(-5.0, 4.0, 10.0, 30000.0, 30000.0, 30000.0);
```

This defines some objects in the virtual world that we're going to ray-trace. Each line is a constructor for a class exported from the `Tracer.dll` library and defined in the header file `Tracer.h`. You'll see the results when you run this final version of the program on your Win32/x86 system.

Next, you'll see a definition for a class named `MyTracer`, which is derived from class `RayTracer`. The only member it defines is a function named `SetPixel()`. This function is called once for each pixel in an image rendered by the ray tracer, to set the pixel to the

proper color in the destination bitmap. The destination could be the screen, a file, or a printer; it makes no difference to the ray-tracing engine.

Listing 3.2 ViewLeft function

```
void CTracerWindow::ViewLeft()
{
    Vector View;

    // Set up view position to the left of the scene
    View[X] = -15.0;
    View[Y] = 0.0;
    View[Z] = 0.0;

    Ray.Trace(&Object1, &Light1,
              PICTURE_WIDTH, PICTURE_HEIGHT, View);
    InvalidateRect(NULL);
}
```

Somewhat further down in the source file, you'll find the definition of function `ViewLeft()`, as shown in [Listing 3.2](#). This function gets called when the user chooses the **From Left** command on the program's View menu; a similar function is provided for each of the other commands on the menu. Each of these commands views the same set of objects from a different angle; the corresponding functions set up a vector to define the viewing position, then call the ray-tracing engine's `Trace` function to generate the image as viewed from that point. The `Trace` function will call back to our `SetPixel()` function to set each pixel in the image to the appropriate color. Finally, we call the Win32 function `InvalidateRect()` to tell the system that part of our window has changed and must be repainted. This will result eventually in a call to our `OnPaint()` function to display the newly generated bitmap.

Now we're finally ready to build the full application:

4. Build the **Ray.mcp** project.

Choose the **Make** command from CodeWarrior's Project menu to build the final version of the ray-tracer application.

5. Run the program.

This time, Ray the Tracer displays the same window, menu, and black box as before. But this time, when you select one of the commands from the View menu, the program displays a ray-traced image of something that might be a weirdly colored water molecule. (It may take some time for the program to compute the image; be patient.)

If you'd like to play around with the code some more, read the comments in the `Tracer.cpp` source code to learn how the object constructors work. You can then try moving objects around, changing their colors, and so forth, to produce an infinite variety of images. If you're a bit more ambitious, you might even want to try adding some new object classes of your own, such as a cylinder, cone, or pyramid. Have fun!



Creating Win32/x86 Applications

A Win32/x86 *application* is a stand-alone, executable program that launches when the user double-clicks it in Windows (or calls it from the DOS command line). Word processors, spreadsheets, paint programs, and utilities are examples of applications.

This chapter includes the following topic:

- [Creating an Application](#)

See also [“Creating Static Libraries for Win32/x86” on page 63](#) and [“Creating Win32/x86 Dynamic Link Libraries” on page 65](#) for information on creating library projects. For more information on projects in general, see the *IDE User Guide*.

Creating an Application

To create a Win32/x86 application, perform the following steps:

1. **Display the New Project dialog box.**

Choose the **New Project** command from the **File** menu. CodeWarrior will display the **New Project** dialog box ([Figure 4.1](#)).

2. **Display the available Win32-x86 project stationery.**

Click the tree control to the left of the **Win32-x86** listing to see the project stationery available to you. [Figure 4.2](#) shows the expanded stationery list.

Creating Win32/x86 Applications

Creating an Application

Figure 4.1 New Project window

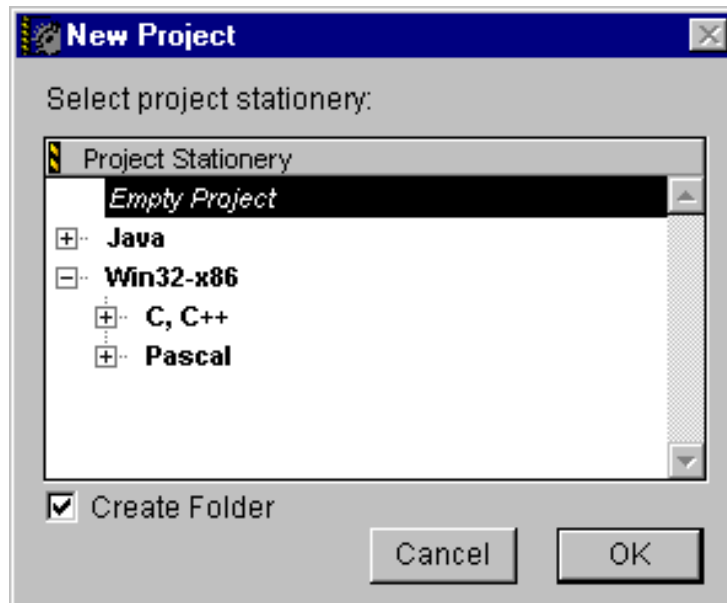
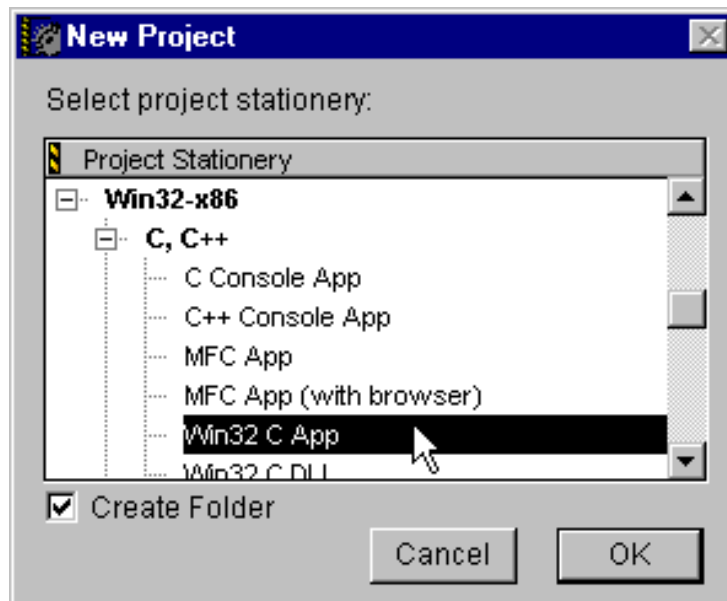


Figure 4.2 Selecting project stationery

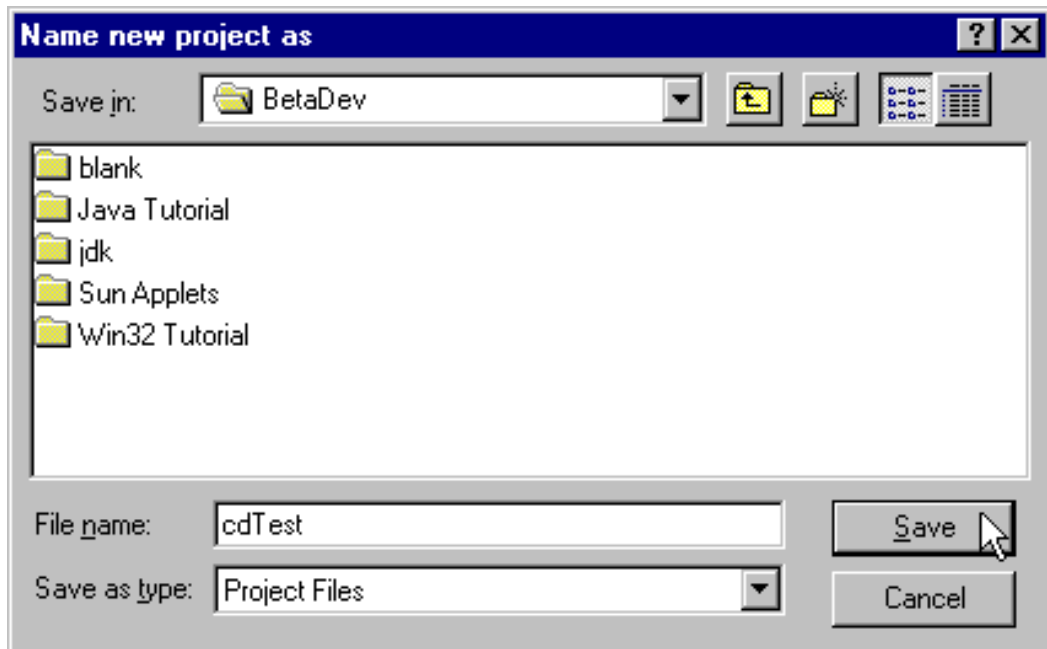


3. **Select your project stationery.**

Click on the line containing the stationery you want, then click **OK**. You'll see the **Name new project as** dialog box ([Figure 4.3](#)).

NOTE: To create a new project without using project stationery, select **Empty Project** in the **New Project** window. This lets you create a project from scratch, but it isn't recommended (particularly for beginners), because of the complexities of including the correct libraries and files and specifying the correct settings.

Figure 4.3 Name new project as dialog



TIP: If you name your project without the `.mcp` extension, CodeWarrior adds `.mcp` to the project name for you. If the **Create Folder** option in the **New Project** dialog box is checked, don't add

Creating Win32/x86 Applications

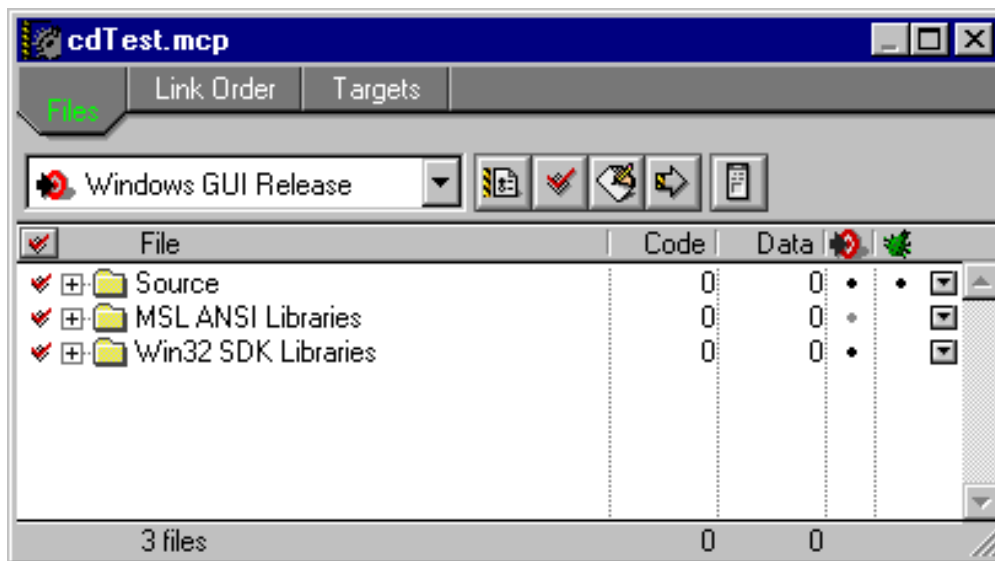
Creating an Application

the `.mcp` extension to your project name. If you do, `.mcp` is added to end of the folder name.

4. Complete the Name new project as dialog.

Navigate to the directory in which you want to place your new project and type the project's name in the box labeled **File name**. When you click the **Save** button, CodeWarrior will create a new project file in the designated directory, with the conventional extension `.mcp` (for "Metrowerks CodeWarrior project"), and will open a project window ([Figure 4.4](#)) containing the appropriate libraries along with a placeholder for the program's source files.

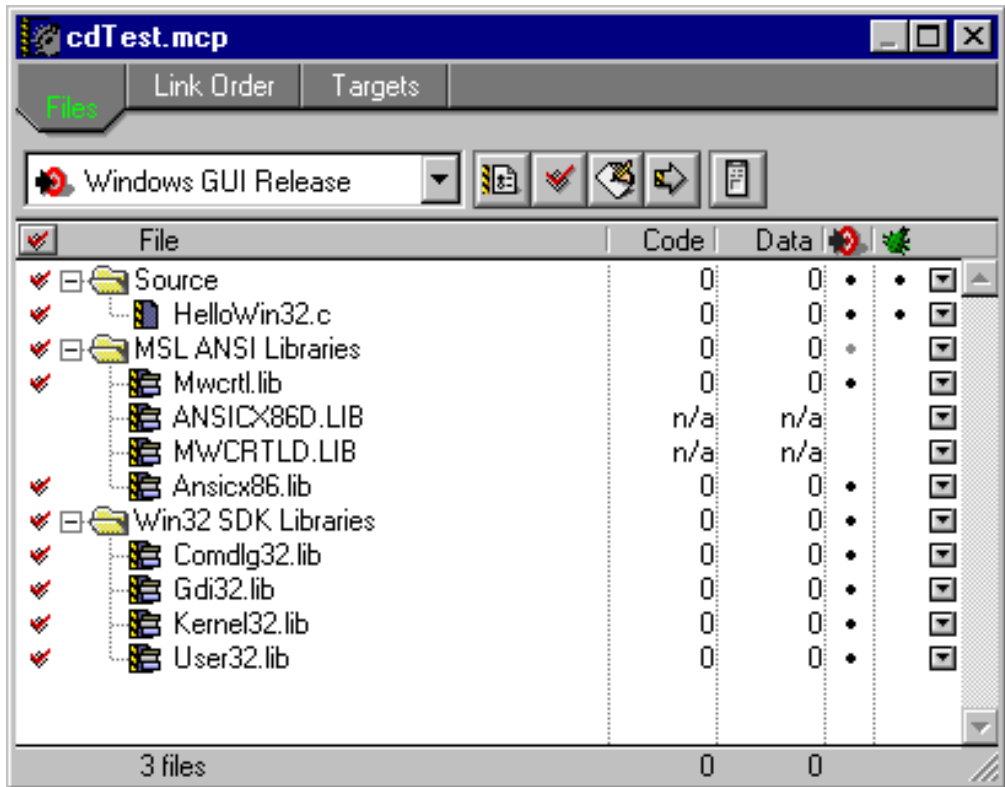
Figure 4.4 Project window



5. Modify the contents of the new project.

Most stationery projects contain source files that are essentially placeholders. Open the folders in your project to view all files ([Figure 4.5](#)). You'll probably want to delete these files and replace them with source files of your own. [Figure 4.6](#) shows the project window with the placeholder `main.c` replaced by a series of source files.

Figure 4.5 Project window showing all files



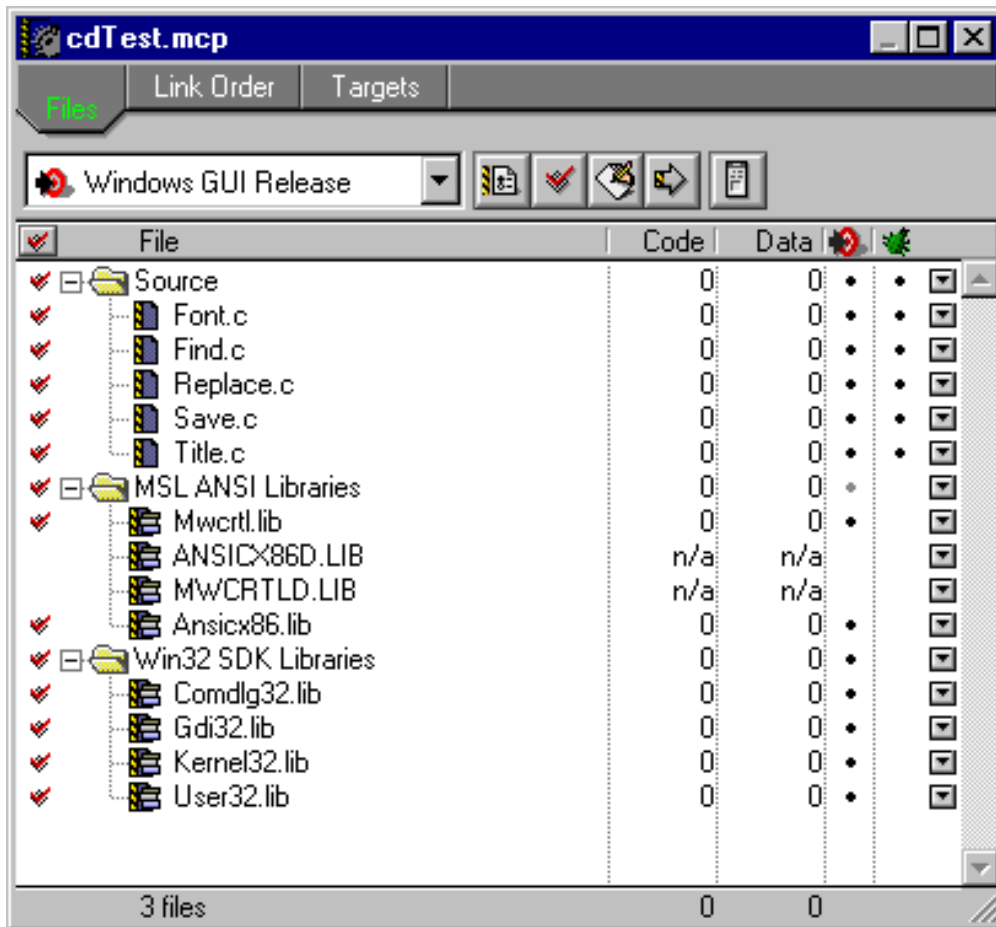
6. Open the Target settings window.

Make sure your project window is active (frontmost) on the screen, then choose the **Settings** command from the **Edit** menu. (The command actually appears on the menu as *Target Settings*, where *Target* is the name of the project's currently selected target. In the project shown in [Figure 4.6](#), for example, the name of the command would be **Windows GUI Release Settings**).

Creating Win32/x86 Applications

Creating an Application

Figure 4.6 Project window with modifications



CodeWarrior displays the Target Settings dialog box ([Figure 4.7](#)), in which you can specify various optional settings for your project. The exact settings you can specify depend on

- the language your source code is written in
- the platform you're compiling for
- the size of your project

and other issues. For Win32/x86 projects, you must specify settings for the target platform, the project type, the compiler, and the linker.

There are various other, optional settings that you can specify as well.

7. Specify target settings.

Click the tree control next to the category name **Target** in the list pane at the left of the window, to display a list of settings panels in that category. Then select **Target Settings**; the window will display the **Target Settings** panel for the project's currently selected target, as shown in [Figure 4.8](#). The **Linker** setting is preset to **Win32 x86 Linker** by the project stationery you selected, but you can edit the target's name or change other settings if you wish.

Figure 4.7 Target settings dialog box

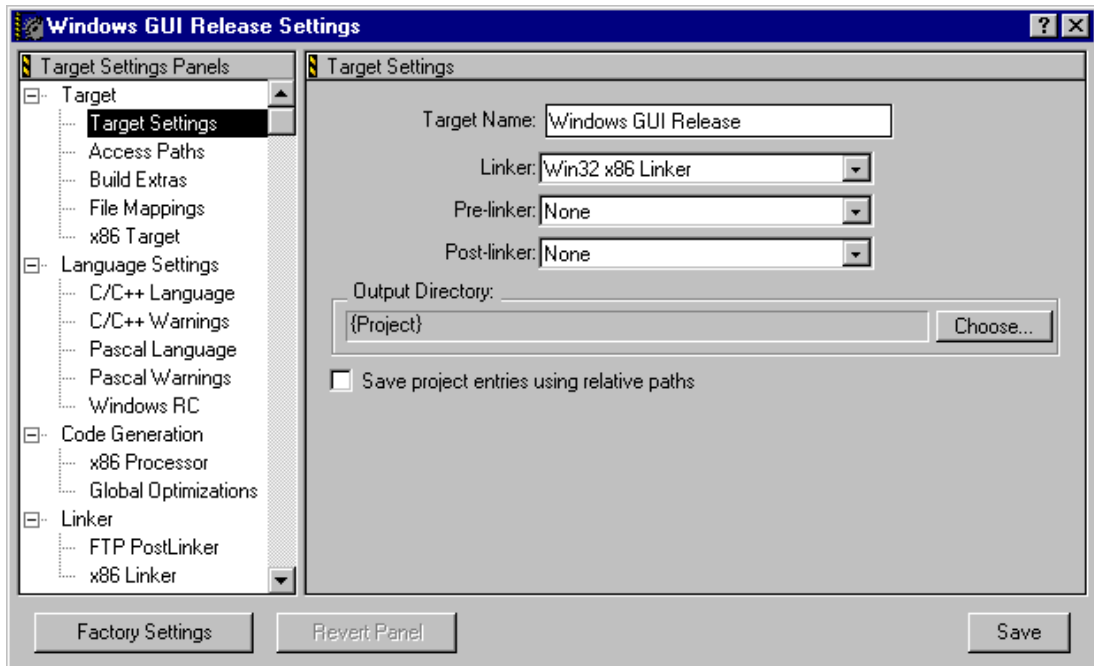
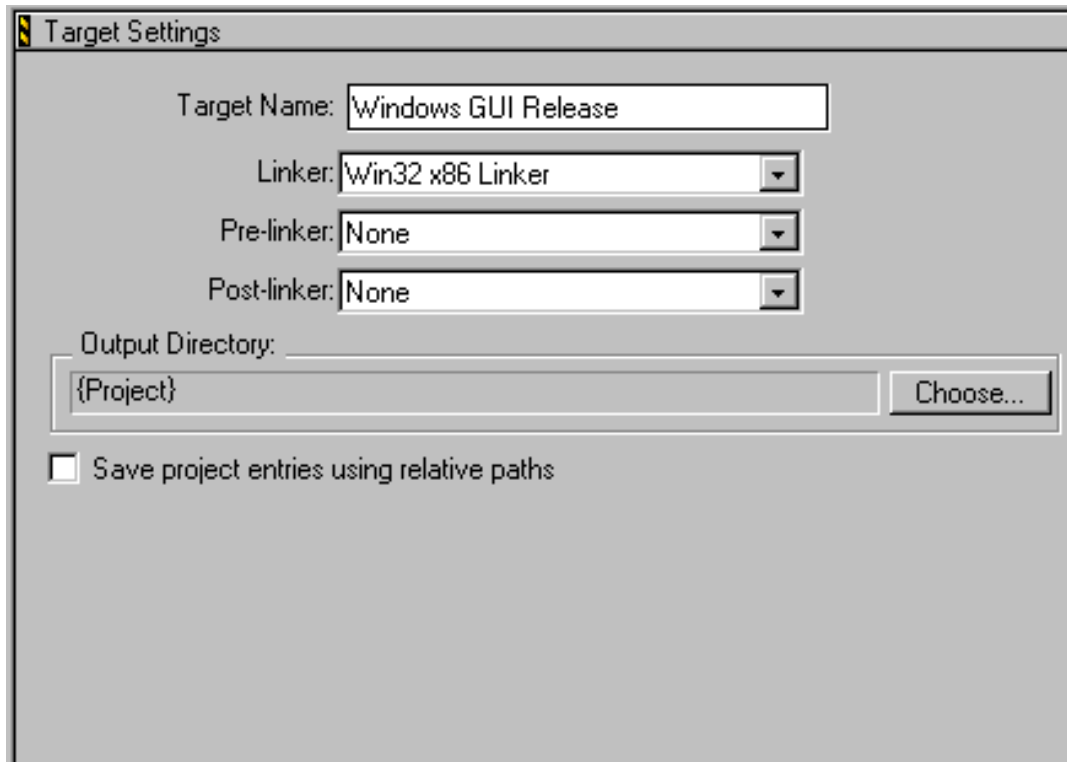


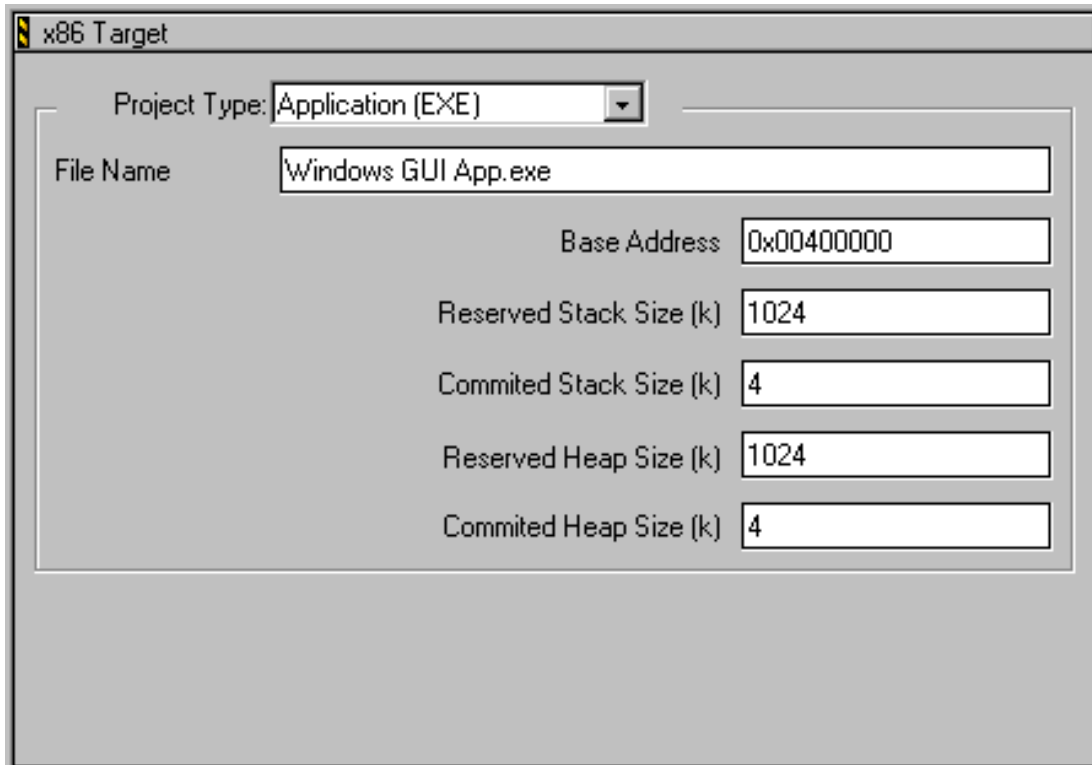
Figure 4.8 Target Settings panel



8. Set the project type.

Click **x86 Target** in the panel list to display the settings panel shown in [Figure 4.9](#). Again, the project type and other default settings are preset for you by the project stationery. For an application project, you should leave the project type set to **Application (EXE)**, but you can modify the output file name and other settings if you wish. (See [“x86 Target” on page 84](#) for more information on the meanings of the various settings.)

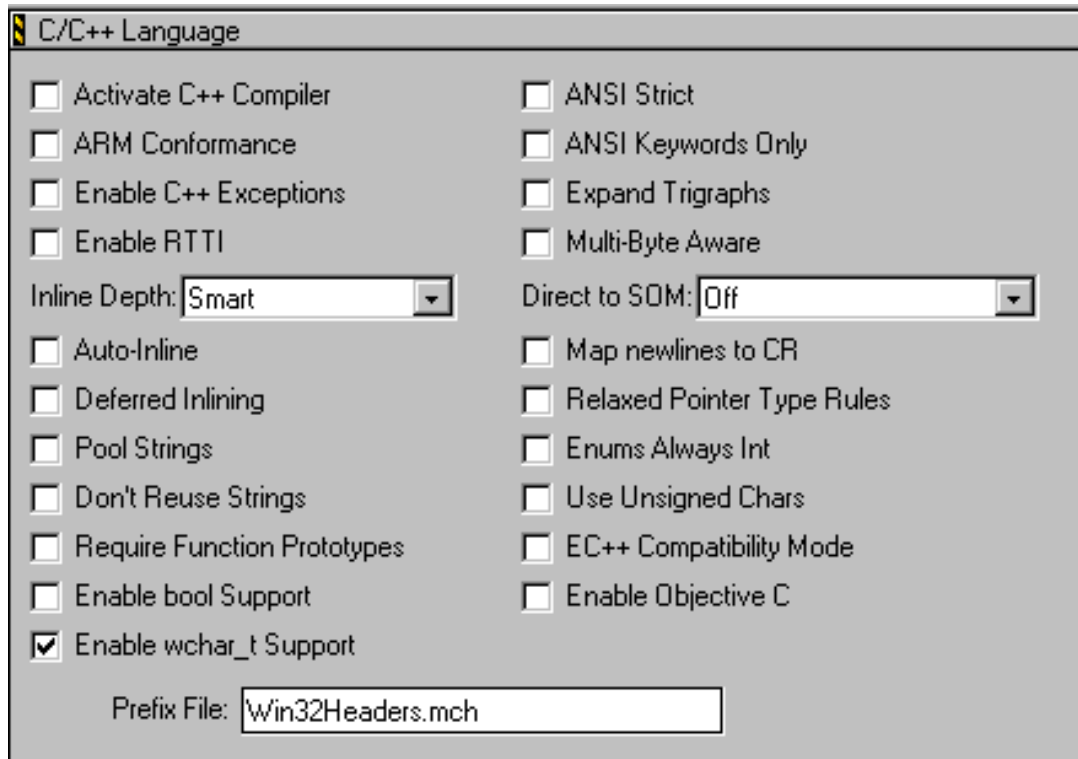
Figure 4.9 x86 Target settings for application projects



9. Specify language settings.

Click the tree control next to the **Language Settings** category in the panel list and select the **C/C++ Language** settings panel ([Figure 4.10](#)). Choose a dialect of C/C++ from the **Source Model** pop-up menu; the rest of the options in the panel will automatically be set to the values most appropriate for that particular dialect. (For instance, the figure shows the settings for **ANSI C**.) Ordinarily, you should just leave these default settings as they are, but you're free to change them if you're sure you know what you're doing. For more information about setting language preferences, see the *C Compilers Reference* and the *Pascal Compilers Reference*.

Figure 4.10 C/C++ Language project settings



10. Specify additional settings.

You can continue to display other project settings panels and specify any settings you wish; for more information on the various panels and settings available, see [“Target Settings for Win32/x86” on page 79](#) as well as the relevant sections of the *IDE User Guide*, the *C Compilers Reference*, and the *Pascal Compilers Reference*. When you’re finished specifying project settings, close the project settings window.

11. Build your project.

After your project is created and its contents and all necessary settings are specified, you’re ready to compile and debug your code. The **Make** command on the Project menu compiles and links your project and, if successful, stores the resulting output file in your project folder, under the name you specified in the **x86 Target** set-

tings panel. (For more information on compiling and linking, see the *IDE User Guide*.) In addition, if you've checked the **SYM Format** option on the **x86 Processor** panel and **Generate SYM File** on the **x86 Linker** panel, CodeWarrior will generate a symbol file, which you'll need for debugging your code with the IDE debugger.

12. Run your application.

Once you've successfully built your application, you can run it in the usual way, such as by double-clicking the **.exe** file's icon or selecting it with the mouse and choosing the **Open** command. If all goes well, congratulations! You've used CodeWarrior to build a running Win32/x86 application. If not, see ["Debugging for Win32/x86" on page 127](#) and the *Debugger User Guide* for information on debugging.

Creating Win32/x86 Applications

Creating an Application



Creating Static Libraries for Win32/x86

This chapter describes the role of static libraries in Win32/x86 projects and how to create them.

Topics in this chapter are:

- [About Static Libraries](#)
- [Creating a Static Library](#)

See also [“Creating Win32/x86 Applications” on page 51](#) and [“Creating Win32/x86 Dynamic Link Libraries” on page 65](#) for information on creating executable applications and dynamic link library (DLL) projects. For more information on projects in general, see the *IDE User Guide*.

About Static Libraries

A *static library* is a collection of functions and data that can be incorporated into an application program (or another library). You can use predefined libraries supplied with CodeWarrior, and you can create your own custom-designed libraries for use in your own projects. Unlike a dynamic link library (DLL), whose contents reside in a separate file and are loaded into memory and linked up with the client program at execution time, the code and data of a static library are included in the client program's project and linked directly into the program at the time it is built.

Creating a Static Library

The steps for creating a static library are essentially the same as those for creating a stand-alone application, but with the following exceptions:

- The **Project Type** in the **x86 Target** settings panel ([Figure 5.1](#)) must be set to **Library (LIB)** instead of **Application (EXE)**.

Figure 5.1 x86 Target panel



- The standard file-name extension for the output file is **.lib** instead of **.exe**.
- After successfully building your static library, you incorporate it into a client program by adding it to the project window before building the program.
- You cannot debug a static library by itself, but can debug it as part of the client program in which it is included.

See [“Creating an Application” on page 51](#) for step-by-step instructions on creating an application project. For details on the various project settings and panels available, see [“Target Settings for Win32/x86” on page 79](#) as well as the relevant sections of the *IDE User Guide*, the *C Compilers Reference*, and the *Pascal Compilers Reference*.



Creating Win32/x86 Dynamic Link Libraries

This chapter tells you how to create, compile, and link dynamic link libraries (DLLs) for use on Win32/x86 systems.

This chapter includes the following topics:

- [About Dynamic Link Libraries](#)
- [A Primer on Exported Names](#)
- [DLL Initialization](#)
- [Creating a Dynamic Link Library](#)

See also [“Creating Win32/x86 Applications” on page 51](#) and [“Creating Static Libraries for Win32/x86” on page 63](#) for information on creating executable applications and static library projects. For more information on projects in general, see the *IDE User Guide*.

About Dynamic Link Libraries

A *dynamic link library* (DLL) is a collection of functions and data that can be shared by application programs (or other libraries) running on a Win32/x86 system. Unlike a static library, whose contents are incorporated directly into the client application, the code and data of a DLL reside in a separate file, formatted very much like an executable application file itself. Because the DLL's contents are loaded into memory and linked to the client application dynamically at execution time, the DLL file must be available on the target machine at the time the application is run.

When you build a DLL, the linker creates two files (where *lib_name* is the name of your DLL project):

- ***lib_name .dll***: The DLL file itself, containing all of the library's code and data.
- ***lib_name.lib***: A static library for linking client programs (applications or even other DLLs) to the DLL. Adding this file to a project makes the contents of the DLL available within that project.

A Primer on Exported Names

DLLs *export* the names of functions or variables to make them accessible to client programs. For compatibility, CodeWarrior's compiler and linker follow conventions established by Microsoft for such exported names. This section describes Microsoft's (admittedly somewhat confusing) naming conventions; it does not attempt to explain the reasons behind them. The topics in this section are:

- [Exporting Names](#)
- [Importing Names](#)
- [Name Transformations](#)

Exporting Names

There are two ways to export a name from a DLL for use by client programs:

- Directly in your source code, by preceding its declaration with the specifier `__declspec(dllexport)`. This specifier exports the name whose declaration immediately follows it in the source code:

```
__declspec(dllexport)
void AnExportedFunction();

__declspec(dllexport)
int AnExportedVariable;
```

For more information, see [“Calling Conventions for Win32/x86” on page 104](#) and the *C Compiler Reference*.

TIP: This is generally the best method to use when building a DLL to be used only by one of your own programs.

- With an `/EXPORT` command in a linker command file. The use of such a command file is optional, and is specified by the **Command File** setting in the **x86 Linker** project settings panel; see [“x86 Linker” on page 92](#) for more information. The `/EXPORT` command has the following syntax:

`/EXPORT: ext_name [=int_name] [ ,@ordinal] [ ,DATA]`

where

- `ext_name` is the external name seen by the client program.
- `int_name` is the C/C++ name used internally in the DLL’s source code, if different from the external name.
- The `@ordinal` option allows you to export a function or variable by number, rather than by name. This feature is an artifact of 16-bit Windows, and generally should not be used in Win32 code.
- The optional keyword `DATA` indicates that the exported name is that of a variable rather than a function.

`/EXPORT:` also supports `[ ,NONAME]` which indicates that a symbol should only be exported by ordinal value. Its name will not appear in the DLL.

`/EXPORT:*` indicates that all external symbols should be exported under their own name.

TIP: This method of exporting names is a bit more flexible than using the `__declspec(dllexport)` specifier, since it allows you to export a function or variable under a different name from the one by which it’s known in the DLL’s own source code. For this reason, this method is often better if you’re creating a DLL that is to be used by existing software: you can use the `/EXPORT` command to export the name in exactly the form the client program expects.

WARNING! Be careful when exporting variables from a DLL that may need to reference those variables. You may need to use two header files to resolve the issue.

Importing Names

The easiest way to import a function into a program from an external DLL is simply to include the static library *lib_name.lib* in your project, and then declare the imported name in your source code with the keyword `extern`. No other special action is needed. However, this method applies only to functions and cannot be used to import variables.

A more general method is to use the specifier `__declspec(dllimport)`, which imports the name whose declaration immediately follows it in the source code:

```
__declspec(dllimport)
void AnImportedFunction();

__declspec(dllimport)
int AnImportedVariable;
```

For more information, see [“Calling Conventions for Win32/x86” on page 104](#) and the *C Compilers Reference*.

TIP: This method is the recommended way of importing all external function and variable names. Not only can it be used with variables as well as functions, it is also slightly more efficient even for functions, because it saves one jump instruction at every call.

Name Transformations

A function or variable exported from a DLL can have as many as four different names, used for different purposes and in different contexts:

- The *C/C++ name* declared in the DLL’s own source code.

- A *linker name*, produced by the compiler for the linker to resolve. The linker name is derived mechanically from the C/C++ name according to the rules described below under [“Linker names.”](#)
- An *external name*, used to link the DLL and its client program dynamically at load time. You specify the external name when you export it from your DLL, as described under [“Exporting Names” on page 66](#); it may or may not be the same as the C/C++ name used in the code of the DLL itself.
- A *library name*, placed in the static library (*lib_name.lib*) for client programs to import.

Linker names

The linker name of an exported function or variable is derived from your original C/C++ name, modified (“decorated”) as follows:

- For C names, an underscore character (`_`) is prefixed to the original name: for example, the name

`CName`

becomes

`_CName`

If you specify the `__stdcall` calling convention, the characters `@nnn` are also appended, where *nnn* is the number of bytes of parameters the function takes: for example,

`_CName@8`

For more information on the `__stdcall` specifier, see [“Calling Conventions for Win32/x86” on page 104](#) and *C Compilers Reference*.

- C names with the `__fastcall` calling convention are prefixed with an '@' (at sign) and suffixed with an '@' and the number of bytes in the parameter list.
- For C++ names, a “mangled” name is generated, allowing the linker to distinguish among different versions of an overloaded function or operator. The mangled name starts with an question mark, followed by the original C++ name or a special form indicating an operator, followed by the encoded forms of the return type, template arguments, throw types, and function parameters, for example:

Creating Win32/x86 Dynamic Link Libraries

A Primer on Exported Names

?_DefaultConstructor@CBase1@@SAXXZ

??_7SplitDerived@@6B@External and library names

The external and library names of an exported function or variable depend on the method you use to export it:

- When you use the `__declspec(dllexport)` declaration specifier to export a function or variable, its linker name (described in the preceding section) also becomes its library name. The external name is then derived as follows:
 - If the name is not a mangled name and is not declared with the `__stdcall` specifier, the external name is simply the undecorated C/C++ name, with no leading underscore added.
 - If the name is mangled or is declared with the specifier `__stdcall`, the external name is the same as the linker name.
- When you use the linker command

`/EXPORT: ext_name`

the external name is exactly as specified in the `/EXPORT` command. The linker then searches for a linker name in the following order:

- a. The exact name specified.
- b. The specified name with a leading underscore added.
- c. Any mangled name or `__stdcall` name that could have been derived from the name specified. In this case, there must be one and only one match among all names in the link.

The linker name found becomes the library name.

- When you use the linker command

`/EXPORT: ext_name=int_name`

the external name is exactly as specified in the `/EXPORT` command. The linker then searches for a linker name as described in the preceding case, but basing the search on the specified internal name (*int_name*) instead of the external name (*ext_name*). In the linker name found, the internal name

int_name is replaced by *ext_name* to arrive at the final library name.

An example may help make all this clear. Suppose you have a C++ function named `Fiddle`, taking 12 bytes of parameters, that is declared with the `__stdcall` modifier and then exported with the linker command

```
/EXPORT: Fiddle=Fiddle
```

This will result in the following names:

- The C/C++ name is `Fiddle`
- The linker name is `_Fiddle@12`
- The external name is `Fiddle`
- The library name is `_Fiddle@12`

NOTE: In all three of the export cases listed above, an additional library name is created. This additional name is formed by adding the prefix `__imp_` to the library name described above. For functions, both the ordinary library name and the one with the `__imp_` prefix are placed in the library; for variables, only the `__imp_` name is used. This ensures that only variables declared with the `__declspec(dllimport)` modifier will match the library name.

DLL Initialization

A Win32 DLL can define an optional entry point. If it does, the entry-point function will be called whenever a process or thread attaches to the DLL or when an attached process or thread detaches or terminates. The entry-point function must be declared with the `__stdcall` specifier (see [“Calling Conventions for Win32/x86” on page 104](#) and the *C Compilers Reference*) and must have the following prototype:

```
BOOL WINAPI  
DllEntry  
(HINSTANCE instDLL,
```

Creating Win32/x86 Dynamic Link Libraries

DLL Initialization

```
        DWORD      fdwReason,  
        LPVOID      lpvReserved);
```

where

- `instDLL` is the module instance handle for the DLL
- `fdwReason` is one of the following constants, indicating the reason the entry point is being called:

```
DLL_PROCESS_ATTACH  
DLL_PROCESS_DETACH  
DLL_THREAD_ATTACH  
DLL_THREAD_DETACH
```

- `lpvReserved` is not currently used

The function's boolean return value indicates success or failure.

The entry point for a DLL is specified by the **Entry Point Usage** setting on the **x86 Linker** project settings panel (described in detail under [“x86 Linker” on page 92](#)). This setting can have the following values:

- **None:** The DLL has no entry point.
- **Default:** The entry point is the standard library function `_DllMainCRTStartup@12` found in the `MWCRtl.lib` and `MWCRtld.lib` library.
- **User Specified:** The name of the entry-point function is specified in the **Name** box of the **x86 Linker** settings panel.

Any DLL that uses the C/C++ run-time library must use the default entry point in order to initialize the run-time library. In particular, you must use the default entry point if your DLL does any of the following:

- Allocates memory
- Performs file operations
- Uses C++ static objects with constructors

If you don't perform any of these operations and don't need to do any special initialization, you can safely set **Entry Point Usage** to **None**.

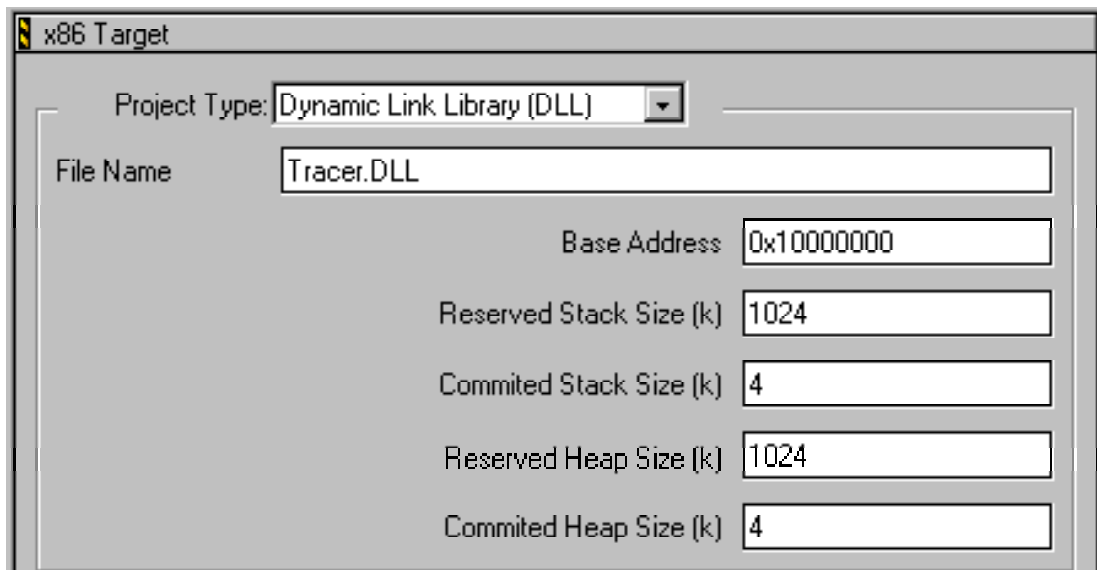
NOTE: To give you a chance to perform additional initialization, the standard entry-point function `_DllMainCRTStartup` calls a user-supplied “hook” function named `_DllMain`. If you use the default entry point, you must supply a `_DllMain`, even if all it does is return `TRUE` to indicate unconditional success.

Creating a Dynamic Link Library

The steps for creating a DLL are essentially the same as those for creating a stand-alone application, but with the following exceptions:

- The **Project Type** in the **x86 Target** settings panel ([Figure 6.1](#)) must be set to **Dynamic Link Library (DLL)** instead of **Application (EXE)**.

Figure 6.1 x86 Target panel



- The standard file-name extension for the output file is `.dll` instead of `.exe`.

Creating Win32/x86 Dynamic Link Libraries

Creating a Dynamic Link Library

- You must specify the **Entry Point Usage** setting in the **x86 Linker** settings panel, as described under [“DLL Initialization” on page 71](#).
- After successfully building your DLL, you must be sure to install it on the hard disk of the target Win32/x86 system, either in the same directory with the client program that will be using it or in any directory mentioned in the `PATH` environment variable.

See [“Creating an Application” on page 51](#) for step-by-step instructions on creating an application project. For details on the various project settings and panels available, see [“Target Settings for Win32/x86” on page 79](#) as well as the relevant sections of the *IDE User Guide*, the *C Compilers Reference*, and the *Pascal Compilers Reference*.

TIP: The easiest way to begin a DLL project with all the appropriate settings and libraries is by selecting the predefined **Win32 C DLL** or **Win32 C++ DLL** stationery from the **New Project** dialog.



Creating Win32 Resources

This chapter tells you how to define Win32 resources and incorporate them into your projects.

Topics in this chapter are:

- [Including Resource Files in a Project](#)
- [Win32 Resource Compiler](#)

Including Resource Files in a Project

There are two kinds of resource file that you can include in a Win32 project:

- A *resource script* (extension `.rc`) is a text file containing descriptions of the desired resources. The CodeWarrior Win32 resource compiler will compile these descriptions into the corresponding resources, which will then be placed in your project's object file.
- A *resource file* (extension `.res`) contains the resources themselves, already created by the resource compiler and ready to be incorporated directly into your object file.

In either case, you can have no more than one resource file in a project. You must either combine all your resource descriptions into one resource script to be passed to the resource compiler, or precompile them all into a single resource file and include that in your project.

NOTE: For now, the only way to create resources for your Win32 projects is to define them in a text-based resource script and compile them with the resource compiler. Future releases of the

CodeWarrior Constructor resource editor will allow you to build Win32 resources interactively on your screen and link them into your projects.

Win32 Resource Compiler

The Win32 resource compiler (RC) is a plug-in tool to the CodeWarrior integrated development environment. It reads standard Win32 resource scripts and compiles them into resources that are then linked into your project's object file.

The CodeWarrior resource compiler accepts resource scripts in the same format as those used by the Microsoft resource compiler. Their syntax is fully documented in the *Microsoft Win32 Reference*, included on the CodeWarrior CD. See in particular the following topics in the *Win32 Reference*:

- “Creating a Resource-Definition File”
- “Single-Line Statements”
- “Multiline Statements”
- “Common Control Parameters”
- “Common Resource Attributes”
- “Sample Resource-Definition File”
- Topics describing individual resource statements (listed in [Table 7.1](#))

Table 7.1 Topic titles for resource script statements

ACCELERATORS Resource	ICON Resource
AUTOCHECKBOX Control	LANGUAGE Statement
AUTORADIOBUTTON Control	LISTBOX Control
AUTO3STATE Control	The Listbox Control Class
BITMAP Resource	LTEXT Control
The Button Control Class	MENU Resource

CAPTION Statement	MENU Statement
CHARACTERISTICS Statement	MENUEX Resource
CHECKBOX Control	MENUITEM Statement
CLASS Statement	MESSAGETABLE Resource
COMBOBOX Control	POPUP Resource
The Combobox Control Class	PUSHBOX Control
CONTROL: General Control	PUSHBUTTON Control
CTEXT Control	RADIOBUTTON Control
CURSOR Resource	RCDATA Resource
DEFPUSHBUTTON Control	RTEXT Control
DIALOG Resource	SCROLLBAR Control
DIALOGEX Resource	The Scrollbar Control Class
The Edit Control Class	STATE3 Control
EDITTEXT Control	The Static Control Class
EXSTYLE Statement	STRINGTABLE Resource
FONT Resource	STYLE Statement
FONT Statement	User-Defined Resource
GROUPBOX Control	VERSION Statement
ICON Control	VERSIONINFO Resource

Resource scripts for the CodeWarrior resource compiler are fully compatible with those accepted by Microsoft's resource compiler, with the following exceptions:

- The CodeWarrior resource compiler handles doubled double quotation marks in the same way as the C compiler, rather than like the Microsoft resource compiler. For example, the Microsoft resource compiler compiles the string

Creating Win32 Resources

Win32 Resource Compiler

`"Hi Mom"`

to

`'Hi Mom'`

In the CodeWarrior resource compiler,

`""Hi Mom"`

compiles to

`'Hi Mom'`

and is combined to become

`'Hi Mom'`

To place a double quotation mark inside a string, use `\"` instead of `""`.

- The CodeWarrior resource compiler ignores directory paths in file names. For example, a resource statement such as

`12 BITMAP .\..\..\res\bits\junk.bmp`

is read simply as

`12 BITMAP junk.bmp`

Use the **Access Paths** settings panel to specify the directory in which to locate files; see the section “Access Paths” in the “Configuring IDE Options” chapter of the *IDE User Guide* for details.



Target Settings for Win32/x86

This chapter discusses each of the settings panels that affect code generation for Win32/x86 development. By modifying the settings for the individual items within a panel you control the compiler, linker, and other aspects of code generation.

Specific details about how the compiler and linker work for Win32/x86 development, such as compiler pragmas, linker symbols and so forth, is found in [“C and C++ for Win32/x86” on page 101](#) and [“Pascal for Win32/x86” on page 111](#).

The sections in this chapter are:

- [Target Settings Overview](#)
- [Settings Panels for Win32/x86](#)

Target Settings Overview

Each target in a CodeWarrior project has its own individual settings. These settings control a variety of features such as compiler options, linker output, error and warning messages, and so forth. You modify these settings through the Target Settings dialog box. This interface is fully explained in the *IDE User Guide*.

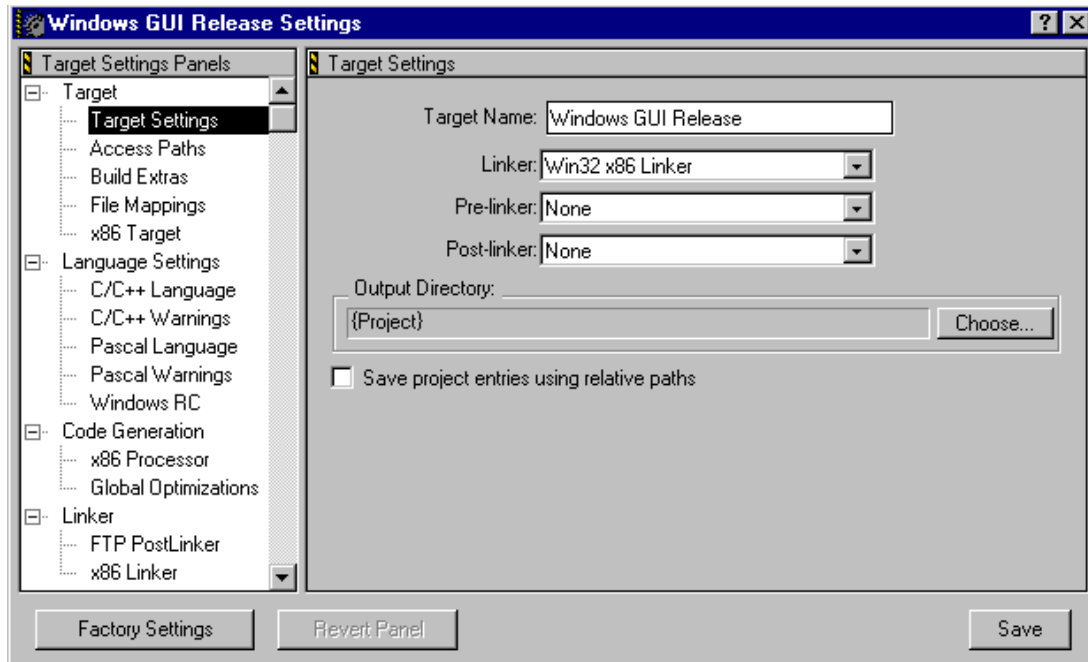
In brief, you control compiler and linker behavior for a particular target by modifying settings in the appropriate settings panels in the Target Settings dialog box. To open any settings panel, choose **Target Settings** from the **Edit** menu, where **Target** is the current target in the CodeWarrior project. Or, go to the Target view of the Project window and double-click the target of interest.

Target Settings for Win32/x86

Target Settings Overview

When you do, the Target Settings dialog box appears, as shown in [Figure 8.1](#).

Figure 8.1 Target Settings dialog box



Select the panel you wish to see from the hierarchical list of panels on the left side of the dialog box. When you do, that panel appears. You can then modify the settings to suit your needs.

When you modify the settings on a panel, you can restore the previous values by using the **Revert Panel** button at the bottom of the dialog box. To restore the settings to the factory defaults, use the **Factory Settings** button at the bottom of the panel.

TIP: Use project stationery when you create a new project. The stationery has all settings in all panels set to reasonable or default values. You can create your own stationery file with your preferred

settings. Modify a new project to suit your needs, then save it in the stationery folder. See the IDE User Guide for details.

Settings Panels for Win32/x86

This section discusses those panels that are specific to Win32/x86 development, and the purpose and effect of each setting. The panels are:

- [Target Settings](#)
- [x86 Target](#)
- [Windows RC](#)
- [x86 Processor](#)
- [x86 Linker](#)
- [Runtime Settings](#)
- [Debugger Settings](#)
- [x86 Exceptions](#)

Settings panels of more general interest are discussed in other CodeWarrior manuals. [Table 8.1](#) lists several panels and where you can find information about them.

Table 8.1 **Where to find information on other settings panels**

Panel	Manual
Access Paths	<i>IDE User Guide</i>
Build Extras	<i>IDE User Guide</i>
File Mappings	<i>IDE User Guide</i>
C/C++ Language	<i>C Compilers Reference</i>
C/C++ Warnings	<i>C Compilers Reference</i>
Pascal Language	<i>Pascal Compilers Reference</i>
Pascal Warnings	<i>Pascal Compilers Reference</i>

Target Settings for Win32/x86

Settings Panels for Win32/x86

Panel	Manual
Global Optimizations	<i>IDE User Guide</i>
Custom Keywords	<i>IDE User Guide</i>

Target Settings

The **Target Settings** *dialog box* contains a **Target Settings** *panel*. The dialog box and the panel are not the same. The dialog box displays all panels, one at a time. The **Target Settings** *panel* is one of those panels.

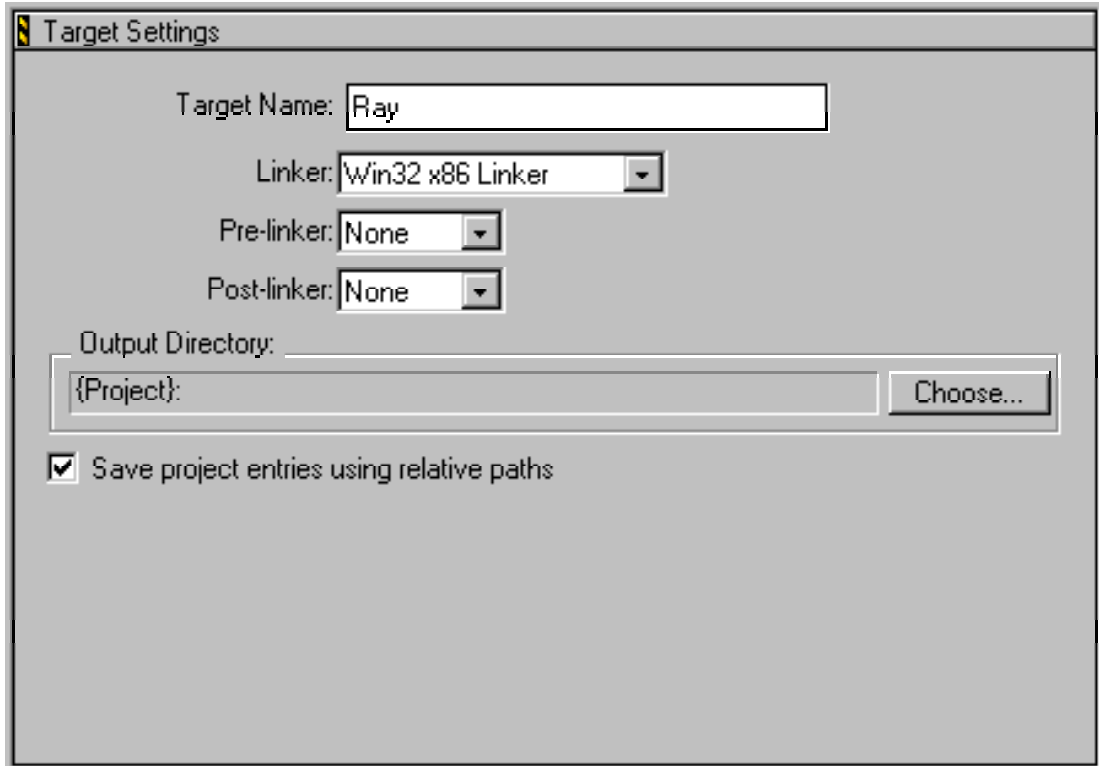
The **Target Settings** panel, shown in [Figure 8.2](#), is perhaps the most important panel in CodeWarrior. This is the panel where you pick your target. When you select a linker in the **Target Settings** panel, you specify the target operating system and/or chip. The other panels listed in the Settings dialog box will change to reflect your choice.

Because the linker choice affects the visibility of other related panels, you must set your target first before you can specify other target-specific options like compiler and linker settings.

The **Target Settings** panel is not the same as the [x86 Target](#) panel. You specify the target in the **Target Settings** panel. You set other target-specific options in the [x86 Target](#) panel.

The items in this panel are:

Target Name	Post-Linker
Linker	Output Directory
Pre-Linker	Save Project Entries Using Relative Paths

Figure 8.2 The Target Settings panel**NOTE: Target Name**

Use the **Target Name** text field to set or change the name of a target. When you use the Targets view in the Project window, you will see the name that you have set.

The name you set here is *not* the name of your final output file. It is the name you assign to the target for your personal use. The name of the final output file is set in the [x86 Target](#) panel.

Linker

Choose a linker from the items listed in the **Linker** pop-up menu. For Win32/x86, use **Win32 x86 Linker** .

Target Settings for Win32/x86

Settings Panels for Win32/x86

Pre-Linker

Some targets have pre-linkers that perform work on object code before it is linked. There is no pre-linker for Win32/x86 development.

Post-Linker

Some targets have post-linkers that perform additional work (such as object code format conversion) on the final executable. There is no post-linker for Win32/x86 development.

Output Directory

This is the directory where your final linked output file will be placed. The default location is the directory that contains your project file. Click the **Choose** button to specify another directory.

Save Project Entries Using Relative Paths

To add two or more files with the same name to a project, select this option. When this option is off, each project entry must have a unique name.

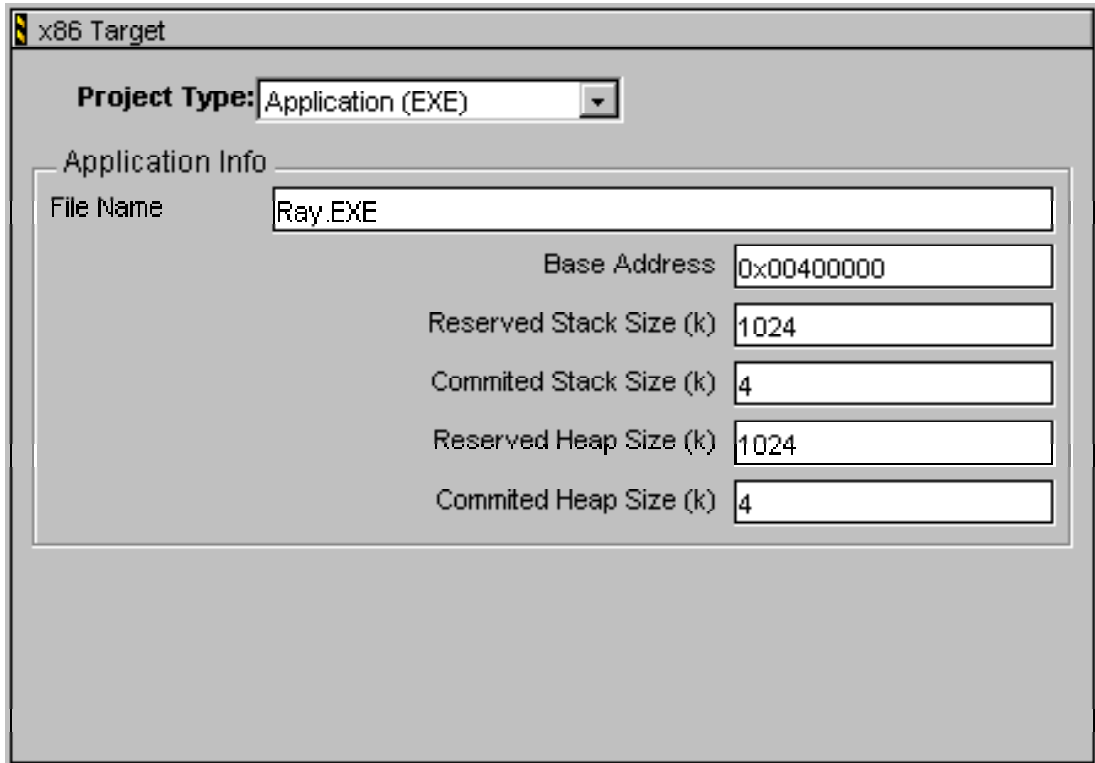
When this option is selected, the IDE includes information about the path used to access the file as well as the file name when it stores information about the file. When searching for a file, the IDE combines **Access Path** settings with the path settings it includes for each project entry.

When this option is off, the IDE only records information about each project entry's file name. When searching for a file, the IDE only uses **Access Paths**.

x86 Target

The **x86 Target** panel, shown in [Figure 8.3](#), is where you set the name of your final output file.

The settings you can specify in this panel depend on the type of project you are creating.

Figure 8.3 The x86 Target panel.

The screenshot shows the 'x86 Target' settings panel. At the top, the 'Project Type' is set to 'Application (EXE)'. Below this, the 'Application Info' section contains the following fields:

Field	Value
File Name	Ray.EXE
Base Address	0x00400000
Reserved Stack Size (k)	1024
Committed Stack Size (k)	4
Reserved Heap Size (k)	1024
Committed Heap Size (k)	4

The items in this panel are:

[Project Type](#)

[File Name](#)

[Base Address](#)

[Reserved Stack Size](#)

[Committed Stack Size](#)

[Reserved Heap Size](#)

[Committed Heap Size](#)

Project Type

Specifies the type of output file to generate. These types are:

- **Application (EXE):** Stand-alone application
- **Dynamic Link Library (DLL):** Dynamic link library
- **Library (LIB):** Static library

Target Settings for Win32/x86

Settings Panels for Win32/x86

File Name

Specifies the name of the application or library file to be generated.

Base Address

Specifies the preferred address for loading the application or DLL. The default is

- 0x00400000 for applications
- 0x10000000 for DLLs

Applications and DLLs load faster when loaded at their preferred addresses.

Reserved Stack Size

Specifies the amount of virtual address space reserved for the stack.

Committed Stack Size

Specifies the amount of memory actually allocated to the stack on loading. This amount automatically increases as needed during execution, up to the reserved stack size.

Reserved Heap Size

Specifies the amount of virtual address space reserved for the dynamic memory heap.

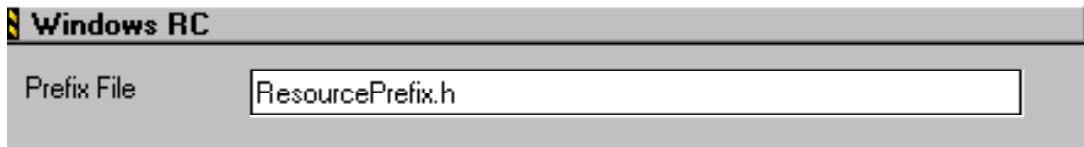
Committed Heap Size

Specifies of memory actually allocated to the heap on loading. This amount automatically increases as needed during execution, up to the reserved heap size. The default is 4 kilobytes.

Windows RC

Use the **Windows RC** panel, shown in [Figure 8.4](#), to configure the Windows Resource Compiler.

Figure 8.4 Windows RC Settings panel



There is only one option in the Windows RC settings panel, **Prefix File**.

Prefix File

Specifies the file to include at the beginning of each resource compiler source file.

x86 Processor

The **x86 Processor** panel, shown in [Figure 8.5](#), is where you control settings related to code generation for the Win32/x86 platform.

The items in this panel are:

[Target Processor](#)

[SYM Format](#)

[Byte Alignment](#)

[CodeView Format](#)

[Exception Handling](#)

[Use MMX/3DNow! Calling Conventions](#)

[MMX](#)

[Inline Intrinsic Functions](#)

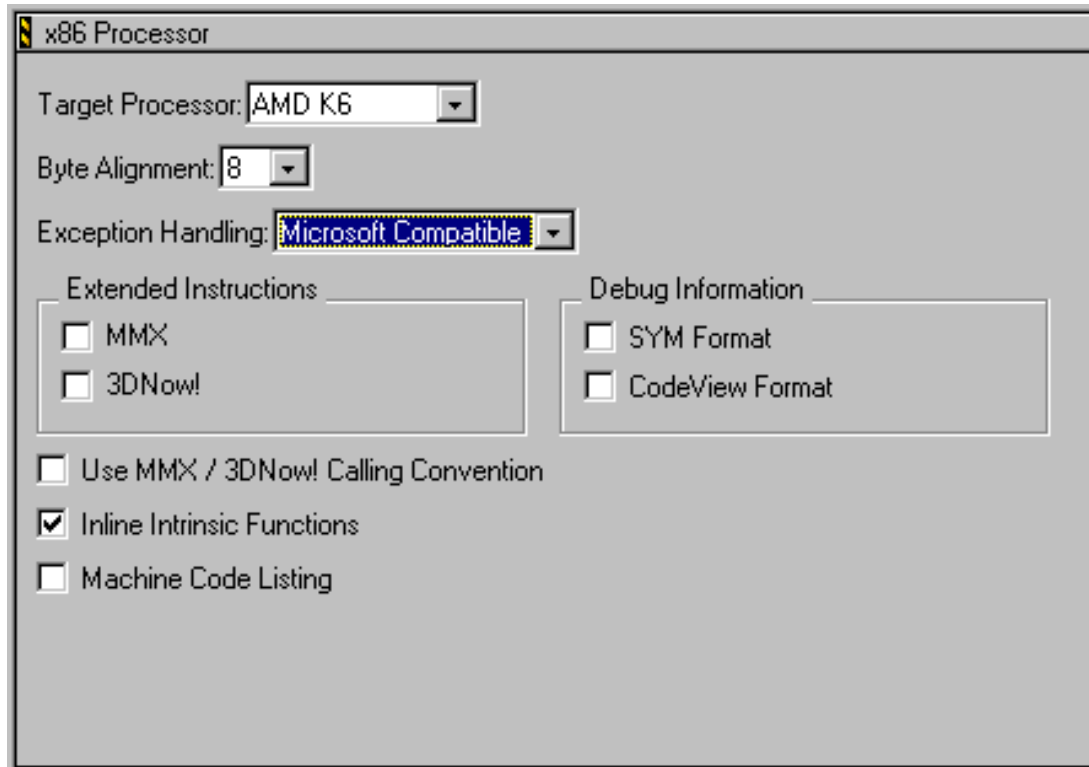
[3DNow!](#)

[Machine Code Listing](#)

Target Settings for Win32/x86

Settings Panels for Win32/x86

Figure 8.5 The x86 Processor panel.



Target Processor

Specifies which processor to generate object code for. The options in this pop-up menu are:

- **Generic 80x86** — generates instructions that execute on Intel 80386, Intel 80486, all Intel Pentium processors, and compatible processors
- **Pentium** — generates instructions that execute on all Intel Pentium and compatible processors
- **Pentium Pro** — generates instructions that execute on Intel Pentium Pro and compatible processors
- **Pentium II** — generates instructions that execute on Intel Pentium II and compatible processors

- **AMD-K6** — generates instructions that execute on Advanced Micro Devices K6 processors

Byte Alignment

Specifies the alignment boundary for structure and class members. Each member is aligned on a multiple of either the specified alignment or the size of the member, whichever is smaller. For example, if you set the byte alignment to 16 and the member's size is 2 bytes, it will be aligned on a 2-byte boundary; if the byte alignment is 4 and the member's size is 8, it will be aligned on a 4-byte boundary. If the byte alignment is set to 1, all members will start on the next available byte, with no padding between members. Setting **Byte Alignment** to 4 or 8 can improve execution speed, but with a penalty in memory usage if structures have wasted space.

Exception Handling

The **Exception Handling** pop-up menu allows you to switch between handling schemes.

The options are:

- **Microsoft Compatible** — allows exceptions to freely cross Microsoft-CodeWarrior boundaries
- **Zero Overhead** — allows you to use CodeWarrior C++ exception handling scheme

MMX

Selecting the **MMX** checkbox enables the generation of MMX instructions.

3DNow!

Selecting the **3DNow!** checkbox enables the generation of code with MMX and 3DNow!™ extensions

To learn more about optimizing for the AMD-K6® processor, refer to [“Optimizing for AMD-K6® Overview” on page 141](#).

Target Settings for Win32/x86

Settings Panels for Win32/x86

SYM Format

Instructs the compiler to generate symbolic debugging information in SYM format.

CodeView Format

Instructs the compiler to generate symbolic debugging information in CodeView format.

NOTE: To generate CodeView symbolics for debugging, you must turn on both the **CodeView Format** setting in the **x86 Processor** panel and the **Generate CV Info** setting in the **x86 Linker** panel.

Use MMX/3DNow! Calling Conventions

Selecting the **Use MMX/3DNow! Calling Conventions** checkbox enables the generation of 3DNow!™ extensions with further optimization for speed.

To learn more about optimizing for the AMD-K6® processor, refer to [“Optimizing for AMD-K6® Overview” on page 141](#).

Inline Intrinsic Functions

The CodeWarrior x86 C/C++ compiler provides intrinsic functions to generate inline x86 instructions. These intrinsic functions execute faster than other functions, because the compiler translates them into inline assembly instructions instead of function calls. A list of the function calls available for inlining is listed in [Figure 8.6](#).

Enable this option to tell the compiler to inline these functions.

Figure 8.6 Functions available for inlining under the x86 C/C++ compiler

```
// memory allocation
void * _alloca (size_t n)
```

```
// block memory operations
void * memmove (void *dest, const void *source, size_t n)
void * memcpy (const void *dest, const void *source, size_t n)
char * strcpy (char *dest, const char *src)
size_t strlen (const char *s)

// floating point math
double sin (double x)
double cos (double x)
double tan (double x)
double atan (double x)
double atan2 (double x)
double acos (double x)
double asin (double x)
double log10 (double x)
double log (double x)
double exp (double x)
double pow (double x)
double sqrt (double x)
double fabs (double x)

// x86 hardware operations
void _outp (unsigned short port, unsigned char val)
void _outpw (unsigned short port, unsigned short val)
void _outpd (unsigned short port, unsigned long val)
unsigned char _inp (unsigned short port)
unsigned short _inpw (unsigned short port)
unsigned long _inpd (unsigned short port)
```

NOTE: Support for intrinsic functions is not part of the ANSI C or C++ standards. They are an extension provided by the CodeWarrior compilers.

WARNING! case is important for intrinsic functions, as is getting the function prototype correct. In the absence of prototypes, the compiler may generate incorrect code for these functions.

Target Settings for Win32/x86

Settings Panels for Win32/x86

Machine Code Listing

Generates a mixed listing of your source files that includes both the original source code and the machine code generated by CodeWarrior. You can use the disassembler to obtain a listing of machine code, but without the accompanying source code. For more information on the disassembler, see “Disassembling Source Code” in the “Compiling and Linking” chapter of the *IDE User Guide*.

x86 Linker

The **x86 Linker** panel, shown in [Figure 8.7](#), is where you control settings related to linking your object code into final form, be it executable, library, or other type of code.

These items in this panel are:

Entry Point Usage	Name
SubSystem	SubSystemID
UserID	Generate Link Map
Generate SYM File	Command File
Generate CV Info	

Figure 8.7 The x86 Linker panel.

x86 Linker

Entry Point

Entry Point Usage: **Default**

Name:

SubSystem & IDs

SubSystem: **Windows CUI**

SubSystem ID:

User ID:

☐ Generate Link Map ☐ Generate SYM File

Command File: ☒ Generate CV Info

Entry Point Usage

Specifies the name of the project's main entry point:

- **Default** — the usual case. The linker will set the entry point automatically, depending on the type of project ([Table 8.2](#)).
- **User Specified** — you supply the entry-point name by typing it into the text box labeled **Name**.

WARNING! Use this option only in projects that don't use C or C++ run-time library functions; otherwise, the run-time library will not be initialized and library functions may fail.

- **None** — no entry point is defined. Use this option only for DLLs that have no initial entry point.

Target Settings for Win32/x86

Settings Panels for Win32/x86

Table 8.2 **Default entry points**

For this project type...	The linker sets the entry point to this symbol
Windows GUI	<code>_WinMainCRTStartup</code>
Console	<code>_mainCRTStartup</code>
DLL	<code>_DllMainCRTStartup@12</code>

Name

Specifies the name of the symbol that provides the entry point for the project's program. This option is only available if **User Specified** is selected from the **Entry Point Usage** pop-up menu.

SubSystem

Select a setting from the **SubSystem** pop-up menu as follows:

- **Unknown** to let the linker determine the subsystem by looking for certain standard main routines:
 - If `_main` is defined, the subsystem is **Windows CUI**.
 - If `_WinMain@16` or `_wWinMain@16` is defined, the subsystem is **Windows GUI**.
- **Native** for drivers and system executables
- **Windows GUI** for applications using the Win32 graphical user interface
- **Windows CUI** for Win32 console applications

SubSystemID

Identifies a specific version of that subsystem. The value of this setting is a floating-point number, with the integer part giving the major ID and the fractional part the minor ID:

- For Windows NT or Windows 3.1, set the subsystem ID to 3.10.

For other subsystems, the subsystem ID is undefined.

UserID

Specifies a version number for your application or library, which will appear in the header of your project's output file. You can set it to any floating-point value you wish.

Generate Link Map

Generates a link map listing the items the linker processed.

Generate SYM File

Generates a SYM file containing symbolic debugging information in SYM format.

Command File

Specifies the name of the linker command file, an optional text file containing additional commands for controlling the operation of the linker.

Generate CV Info

Stores symbolic debugging information in the object file (whether an executable application or a DLL) in CodeView format.

NOTE: To generate symbolic debugging information in either SYM or CodeView format, you must turn on the corresponding settings in both the **x86 Processor** and **x86 Linker** project settings panels.

Runtime Settings

The **Runtime Settings** panel presents options for configuring the runtime environment that your project will run in while it is under the debugger's control.

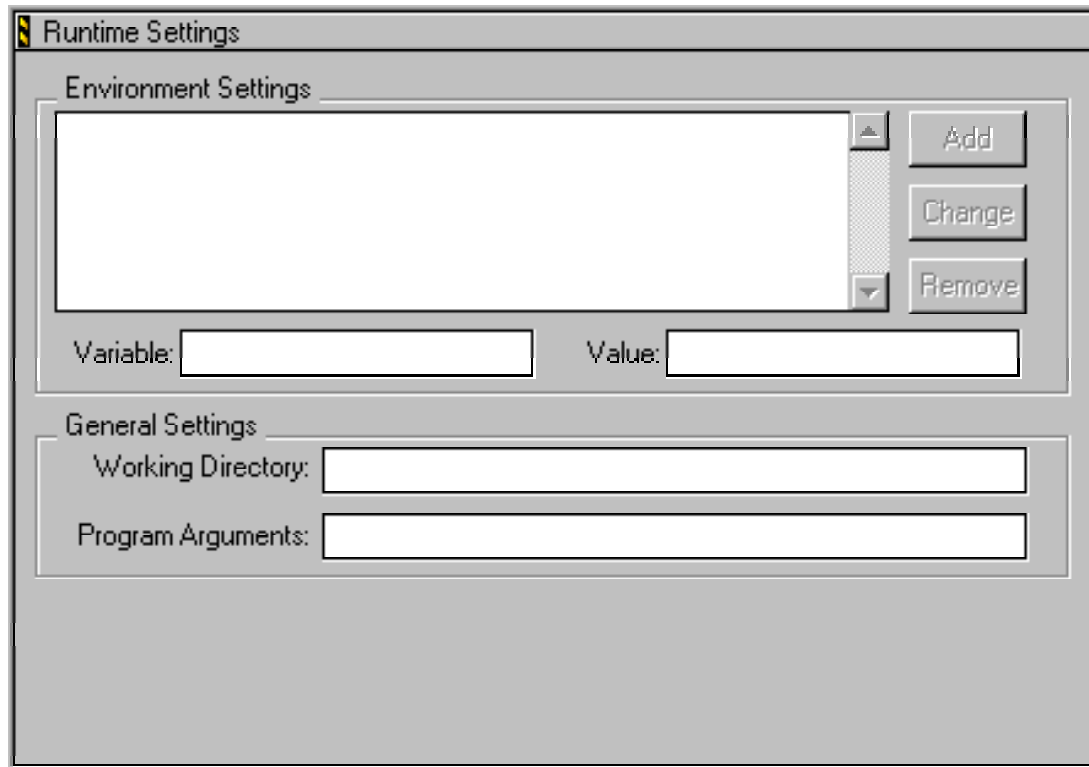
Environment variables listed in this panel are created when the project is launched and removed when the project finishes execut-

Target Settings for Win32/x86

Settings Panels for Win32/x86

ing. The IDE ensures that these environment variables have no effect on previously defined or global environment variables.

Figure 8.8 Runtime Settings panel



The options in this panel are:

[Add](#)

[Change](#)

[Remove](#)

[Variable](#)

[Value](#)

[Working Directory](#)

[Program Arguments](#)

Add

Adds the environment variable in the **Variable** field to the list of environment variables and assigns it the value in the **Value** field.

Change

Replaces the value assigned to the variable selected in the list with the value entered in the **Value** field.

Remove

Deletes the selected items in the environment variable list.

Variable

Specifies the name of an environment variable to add or change in the list of environment variables.

Value

Specifies the value to assign to the variable specified in the **Variable** field.

Working Directory

Specifies the default directory that the project will have access to.

Program Arguments

Specifies command-line text to pass to the project when it is launched by the IDE's debugger.

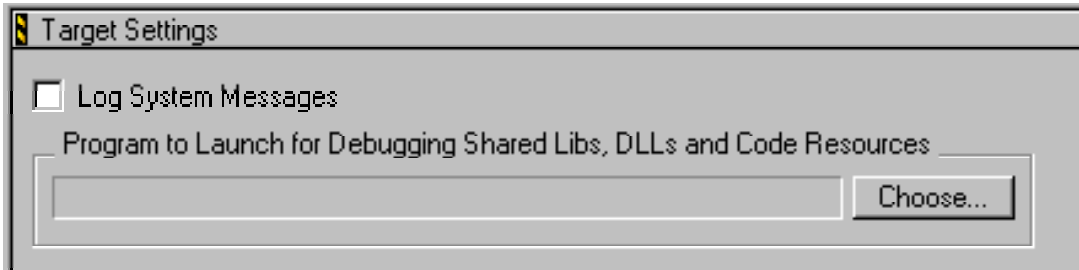
Debugger Settings

The **Debugger Settings** panel presents options for selecting the tools to use for debugging your project.

Target Settings for Win32/x86

Settings Panels for Win32/x86

Figure 8.9 Debugger Settings panel



The options in this panel are:

[Log System Messages](#)

[Program to Launch for Debugging](#)

Log System Messages

Controls the display of messages in the debugger's log window. If selected, messages about the actions the debugger takes while controlling your application appear in the log window.

NOTE: This feature is only available when debugging on Windows.

Program to Launch for Debugging

Launches a program before running the project under the debugger's control. Click the **Choose** button to select a program. This option is most useful when debugging a piece of software that acts as a plugin or extension to another program.

x86 Exceptions

The **x86 Exceptions** panel, shown in [8.10](#), lists all the exceptions that the debugger is able to catch. If you want the debugger to catch all the exceptions, you should select the checkboxes of all the options in this panel. However, if you prefer to handle some of the exceptions, leave the checkboxes of those exceptions unselected.

Figure 8.10 x86 Exceptions panel

x86 Exceptions

Exception Handling (place a check on the exceptions to always catch)

☐ 0x40010005 Control-C	☐ 0xC000008F Float Inexact
☐ 0x40010008 Control-Break	☐ 0xC0000090 Float Invalid Op
☐ 0x80000002 Data Misaligned	☐ 0xC0000092 Float Stack
☐ 0xC0000005 Access Violation	☐ 0xC0000091 Float Overflow
☐ 0xC0000006 In Page Error	☐ 0xC0000093 Float Underflow
☐ 0xC0000017 No Memory	☐ 0xC0000094 Int Div by Zero
☐ 0xC000001D Illegal Instruction	☐ 0xC0000095 Int Overflow
☐ 0xC0000025 Noncontinuable	☐ 0xC0000096 Privileged Instr
☐ 0xC0000026 Invalid Disposition	☐ 0xC00000FD Stack Overflow
☐ 0xC000008C Array Bounds	☐ 0xC0000135 DLL Not Found
☐ 0xC000008D Float Denormal	☐ 0xC0000142 DLL Init Failed
☐ 0xC000008E Float Div by Zero	☐ 0xE06d7363 C++ Exception

Target Settings for Win32/x86

Settings Panels for Win32/x86



C and C++ for Win32/x86

This chapter describes both the CodeWarrior back-end compiler and linker for Win32/x86, including support for inline assembly in the compiler.

The sections in this chapter are:

- [Number Formats for Win32/x86](#)
- [Calling Conventions for Win32/x86](#)
- [Variable Allocation for Win32/x86](#)
- [Register Variables for Win32/x86](#)
- [Optimizing Code for Win32/x86](#)
- [Linker Issues for Win32/x86](#)

However, this chapter does *not* discuss several important topics with respect to code generation, including front-end compiler issues, compiler and linker errors, controlling the size of C++ code, and so forth. These topics are covered in other CodeWarrior documentation as outlined in [Table 9.1](#).

Table 9.1 Other compiler/linker documentation

For this topic...	See...
how CodeWarrior implements the C/C++ language	<i>C Compilers Reference</i> generally
using C/C++ Language and C/C++ Warnings settings panels	<i>C Compilers Reference</i> , “Setting C/C++ Compiler Options” chapter

For this topic...	See...
controlling the size of C++ code	<i>C Compilers Reference</i> , “C++ and Embedded Systems” chapter
using compiler pragmas	<i>C Compilers Reference</i> , “Pragmas and Symbols” chapter
initiating a build, controlling which files are compiled, handling error reports	<i>IDE User Guide</i> , “Compiling and Linking” chapter
information about a particular error	<i>Error Reference</i>
Win32/x86 assembler	<i>Assembler Guide</i>

NOTE: Some of the items discussed in this chapter may actually be implemented in the front-end compiler. However, it really doesn't matter whether the actual implementation of a feature occurs in the front-end or back-end compiler. From the programmer's point of view, it is all one compiler.

Number Formats for Win32/x86

This section describes how the CodeWarrior C/C++ compilers implement integer and floating-point types for Win32/x86 processors. You can also read `limits.h` for more information on integer types, and `float.h` for more information on floating-point types.

The topics in this section are:

- [Win32/x86 Integer Formats](#)
- [Win32/x86 Floating-Point Formats](#)

Win32/x86 Integer Formats

The Win32/x86 back-end compiler does not let you change the sizes of integers. The size of a `short int` is always 2 bytes and the size of `int` or `long int` is always 4 bytes.

[Table 9.2](#) shows the size and range of the integer types for the Win32/x86 compiler.

Table 9.2 Win32/x86 Integer Types

For this type	Option setting	Size is	and its range is
<code>bool</code>	n/a	8 bits	true or false
<code>char</code>	UseUnsignedChars is off	8 bits	–128 to 127
	UseUnsignedChars is on	8 bits	0 to 255
<code>signed char</code>	n/a	8 bits	–128 to 127
<code>unsigned char</code>	n/a	8 bits	0 to 255
<code>short</code>	n/a	16 bits	–32,768 to 32,767
<code>unsigned short</code>	n/a	16 bits	0 to 65,535
<code>int</code>	n/a	32 bits	–2,147,483,648 to 2,147,483,647
<code>unsigned int</code>	n/a	32 bits	0 to 4,294,967,295
<code>long</code>	n/a	32 bits	–2,147,483,648 to 2,147,483,647
<code>unsigned long</code>	n/a	32 bits	0 to 4,294,967,295
<code>long long</code>	n/a	64 bits	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
<code>unsigned long long</code>	n/a	64 bits	0 to 18,446,744,073,709,551,615

Win32/x86 Floating-Point Formats

[Table 9.3](#) shows the sizes and ranges of the floating point types for the Win32/x86 compiler.

Table 9.3 Win32/x86 floating point types

Type	Size	Range
float	32 bits	1.17549e-38 to 3.40282e+38
short double	64 bits	2.22507e-308 to 1.79769e+308
double	64 bits	2.22507e-308 to 1.79769e+308
long double	64 bits	2.22507e-308 to 1.79769e+308

Calling Conventions for Win32/x86

This section describes the C/C++ calling conventions for Win32/x86 development.

The Win32/x86 back-end compiler supports four calling conventions:

- default
- standard call
- this call
- fast call

Unless you specify otherwise, the compiler uses the default calling convention. You may choose another calling convention with these declaration specifiers:

- `__stdcall`
- `__thiscall`

NOTE: `__thiscall` is not a supported keyword; it is only available as the default calling convention for C++ methods.

- `__fastcall`

Default calling convention

The default calling convention pushes all parameters onto the stack in right to left order, so the first parameter in the list is on top of stack when the call is made. It expands each parameter to at least 32 bits on the stack and pads structs to an even number of 32 bit long-words.

When the function call returns, the caller removes the parameters from the stack.

The compiler returns a function's value in one of these ways:

- It returns integer and pointer values in the EAX register
- It returns floating point values on the floating point processor stack
- It returns structures and classes by passing an additional parameter with the address of a temporary variable, and pushes that address onto the stack after all explicit parameters

The default calling convention is also known as the `__cdecl` calling convention.

Standard calling convention

The standard calling convention is the same as the default calling convention, except that the *callee* removes parameters from the stack. If you're declaring a function for an API, specify the standard calling convention with `__stdcall`.

This calling convention

The `__thiscall` calling convention is the same as the standard calling convention, except that it passes the `this` pointer in the ECX register.

If you're declaring a non-static member function, the compiler automatically uses the `__stdcall` calling convention unless you explicitly specify the standard calling convention with `__cdecl`.

Fast calling convention

The `__fastcall` calling convention passes the first two arguments that are 4 bytes or smaller in the x86 processor registers ECX and EDI.

Variable Allocation for Win32/x86

(K&R, §A4.3, §A8.3, §A8.6.2) The Win32/x86 back-end compiler places no limits on how large variables can be or how you allocate them.

Register Variables for Win32/x86

(K&R, §A4.1, §A8.1) This section describes how the C/C++ compilers allocate variables to registers.

The Win32/x86 back-end compiler automatically allocates local variables and parameters to registers according to how frequently they're used and how many registers are available. If you're optimizing for speed, the compiler gives preference to variables used in loops.

The Win32/x86 compiler ignores the `register` declaration and decides on its own which variables to place in registers.

The compiler will also use the floating point stack (ST(0) through ST(7)) for local variables.

The Win32/x86 compiler can use these registers for local variables:

- EAX

- EBX
- ECX
- EDX
- ESI
- EDI

Optimizing Code for Win32/x86

This section discusses optimizations that are specific to Win32/x86 development with CodeWarrior. These are:

- [Register Coloring](#)
- [Instruction Scheduling](#)

More optimizations are available through the **Global Optimizations** target settings panel. Refer to the *IDE User Guide* for more information on optimizations and this panel.

Register Coloring

The C/C++ compiler can perform an optimization called *register coloring*. In this optimization, the compiler assigns two or more variables to the same register. It does this if the code does not use the variables at the same time. In this example, the compiler could place *i* and *j* in the same register:

```
short i;  
int j;  
  
for (i=0; i<100; i++) { MyFunc(i); }  
for (j=0; j<100; j++) { MyFunc(j); }
```

However, if a line of code like the one below appears anywhere in the function, the compiler would realize that you are using *i* and *j* at the same time, and place them in different registers.

```
MyFunc (i + j);
```

Register coloring reduces code size and has no effect on execution time.

If register coloring is on while you debug your code, it may appear as though there's something wrong with the variables that share a single register. In the example above, *i* and *j* would always have the same value. When *i* changes, *j* changes in the same way, and vice versa.

Instruction Scheduling

The compiler uses the *instruction scheduling* optimization to increase the speed of execution. This optimization rearranges processor instructions so that the execution of one instruction doesn't hold up the execution of others, whenever possible. This optimization only has an effect on Win32/x86 processors that are able to process more than one instruction at a time. Namely, Intel Pentium (and compatible) processors have multiple execution units that handle more than one type of instruction at a time.

To turn on instruction scheduling, slide the dial on the **Global Optimizations** target settings panel to Level 3 or Level 4. To learn about the **Global Optimizations** panel, see the *IDE User Guide*.

The option you choose from the [Target Processor](#) menu on the [x86 Processor](#) target settings panel controls the scheduling heuristics for a particular processor. For more information about this panel, see ["x86 Processor" on page 87](#).

To turn off register coloring, disable the [SYM Format](#) checkbox in the [x86 Processor](#) settings panel.

Linker Issues for Win32/x86

This section discusses the background information on the Win32/x86 linker and how it works. The topics in this section are:

- [Deadstripping Unused Code and Data](#)

- [Link Order](#)

Deadstripping Unused Code and Data

The Win32/x86 linker deadstrips unused code and data only from files compiled by the CodeWarrior C/C++ compiler. Assembler relocatable files and C/C++ object files built by other compilers are never deadstripped. Deadstripping is particularly useful for C++ programs. Libraries (archives) built with the CodeWarrior C/C++ compiler only contribute the used objects to the linked program. If a library has assembly or other C/C++ compiler built files, only those files that have at least one referenced object contribute to the linked program. Completely unreferenced object files are always ignored.

Link Order

Link order is generally specified in the Link Order view of the Project window. For general information on setting link order, see the *IDE User Guide*.

Regardless of the link order specified in the Link Order view of the Project window, the Win32/x86 linker always processes C/C++ or assembler source files before it processes relocatable files (.obj) or archive files (.lib), which are treated as libraries. Therefore if a symbol is defined in a source file, the linker will use that definition in preference to a definition in a library.



Pascal for Win32/x86

This chapter describes both the CodeWarrior back-end compiler and linker for Win32/x86, including support for inline assembly in the compiler.

The sections in this chapter are:

- [Language Extensions for Win32/x86](#)
- [Number Formats for Win32/x86](#)
- [Calling Conventions for Win32/x86](#)

This chapter does *not* discuss several important topics with respect to code generation, including front-end compiler issues, compiler and linker errors, and so forth. These topics are covered in other CodeWarrior documentation as outlined in [Table 10.1](#).

Table 10.1 Other compiler/linker documentation

For this topic...	See...
how CodeWarrior implements the Pascal and Object Pascal languages	<i>Pascal Compilers Reference</i> generally
using Pascal Language and Pascal Warnings settings panels	<i>Pascal Compilers Reference</i> , “Setting Pascal Compiler Options” chapter
using compiler pragmas	<i>Pascal Compilers Reference</i> , “Pragmas and Symbols” chapter
initiating a build, controlling which files are compiled, handling error reports	<i>IDE User Guide</i> , “Compiling and Linking” chapter

Pascal for Win32/x86

Language Extensions for Win32/x86

For this topic...	See...
information about a particular error	<i>Error Reference</i>
Win32/x86 assembler	<i>Assembler Guide</i>

NOTE: Some of the items discussed in this chapter may actually be implemented in the front-end compiler. However, it really doesn't matter whether the actual implementation of a feature occurs in the front-end or back-end compiler. From the programmer's point of view, it is all one compiler.

Language Extensions for Win32/x86

This section describes the Win32/x86-specific extensions to the Pascal standards found in the CodeWarrior Pascal compiler.

You can disable some of these extensions with options in the **Pascal Language** panel. This panel is described fully in the *Pascal Compilers Reference* manual.

The topic in this section are:

- [Turbo Pascal Support](#)

Turbo Pascal Support

CodeWarrior Pascal supports certain routines from the well-known Turbo Pascal implementation of the Pascal language. This compatibility is activated through the **IO Model** option in the **Pascal Language** settings panel.

Number Formats for Win32/x86

This section describes how the CodeWarrior Pascal compilers implement integer and floating-point types for Win32/x86 processors.

The topics in this section are:

- [Win32/x86 Integer Formats](#)
- [Win32/x86 Floating-Point Formats](#)

Win32/x86 Integer Formats

This section describes how `INTEGER`, `LONGINT` and a few other important atomic data types are implemented for x86 in CodeWarrior Pascal.

INTEGER for x86

`INTEGER`s on x86 platform are 16-bit, 2's-complement integers. The range of the type `INTEGER` is the set of values

$-(\text{MaxINT}+1), -\text{MaxINT}, \dots,$
 $-1, 0, 1, \dots \text{MaxINT}-1, \text{MaxINT}$

that is, -32768 to $+32767$.

LONGINT for x86

For x86 these are 32-bit, 2's-complement integers. The range of the type `LONGINT` is the set of values

$-(\text{MaxLONGINT}+1), -\text{MaxLONGINT}, \dots -1, 0, 1,$
 $\dots \text{MaxLONGINT}-1, \text{MaxLONGINT}$

that is -2147483648 to $+2147483647$.

Arithmetic on `INTEGER` and `LONGINT` operands is done in both 16-bit and 32-bit precision, as follows:

- All integer constants in the range of type `INTEGER` are considered to be of type `INTEGER`. All integer constants in the range of type `LONGINT`, but not in the range of type `INTEGER`, are considered to be of type `LONGINT`.

- When both operands of an operator (or the single operand of a unary operator) are of type `INTEGER`, 16-bit operations are always performed and the result is of type `INTEGER` (truncated to 16 bits if necessary). Similarly, if both operands are of type `LONGINT`, 32-bit operations are always performed and the result is of type `LONGINT`.
- When one operand is of type `LONGINT` and the other is of type `INTEGER`, the `INTEGER` operand is converted to `LONGINT`, 32-bit operations are performed, and the result is of type `LONGINT`. However, if this value is assigned to a variable of type `INTEGER`, it is truncated (see next rule).
- The expression on the right of an assignment statement is evaluated independently of the size of the variable on the left. For example, if variable `longVar` is declared as type `LONGINT`, the statement `longVar := MaxINT+MaxINT` will still cause `INTEGER` overflow. If necessary, the result of the expression is truncated or extended to match the size of the variable on the left.

An important point to remember is that each operator is applied only to its two operands, so, at most, one of those operands is converted. If the expression contains other operands, those are not necessarily converted. For example, in the expression

`oneInt+twoInt+threeInt+oneLongint`

the value of `oneInt` is added to `twoInt` in a 16-bit operation, the result is added to `threeInt` in another 16-bit operation, and the result of that is converted to a `LONGINT` value and added to `oneLongint`. The result of the expression is a `LONGINT`.

The `Ord4` function described in the Pascal Library Reference can be used to convert an `INTEGER` value to a `LONGINT` value.

UNSIGNEDBYTE for x86

The CodeWarrior Pascal compiler implements this type as an 8-bit unsigned integer for the x86 platform. The range of the type `UNSIGNEDBYTE` is the set of values 0 to +255.

This type can be used in expressions with the UNSIGNEDWORD, UNSIGNEDLONG, INTEGER and LONGINT types, but no range checking is available.

UNSIGNEDWORD for x86

The CodeWarrior Pascal compiler implements this type as a 16-bit unsigned integer for x86. The range of the type UNSIGNEDWORD is the set of values

$$0, 1, \dots, 2 * \text{MaxINT}, (2 * \text{MaxINT}) + 1$$

that is, 0 to +65535.

This type can be used in expressions with the UNSIGNEDLONG, INTEGER and LONGINT types, but no range checking is available.

UNSIGNEDLONG for x86

The CodeWarrior Pascal compiler implements this type as a 32-bit unsigned integer for the x86 platform. The range of the type UNSIGNEDLONG is the set of values

$$0, 1, \dots, 2 * \text{MaxLONGINT}, (2 * \text{MaxLONGINT}) + 1$$

that is, 0 to +4294967295.

This type can be used in expressions with the UNSIGNEDWORD, INTEGER and LONGINT types, but no range checking is available.

Win32/x86 Floating-Point Formats

The ANS REAL for x86 type is identical to the SINGLE for x86 type. CodeWarrior Pascal accepts both identifiers and treats them identically.

REAL for x86

For the x86 platform, this type takes up four bytes of storage (32 bits). The range of this type is approximately 1.401298464E-45 to 3.402823466E38 in scientific notation. REAL has 7 or 8 digits of precision.

SINGLE for x86

This type is the same as `REAL` for `x86`.

LONGREAL for x86

For the `x86` platform, the CodeWarrior Pascal compiler implements this data type as an 8-byte double, 64 bits in length. The range of this type is approximately 5.0E-324 to 1.7E308, expressed in scientific notation.

DOUBLE for x86

This type is the same as `LONGREAL`.

Calling Conventions for Win32/x86

This section describes the Pascal calling conventions for Win32/x86 development.

For projects targeting Win32/x86, the CodeWarrior Pascal compiler supports the Microsoft® Win32/x86 calling conventions for `__cdecl`, `__fastcall` and `__stdcall`, plus its own Pascal calling convention.

The information in this section is based on the *Microsoft Visual C++ 2.0 Language Reference*, the fifth volume in the printed Visual C++ documentation set.

Calling convention is selected on a per-procedure and per-function basis, by adding `:__stdcall`, `:__cdecl` or `:__fastcall` to the end of your function or procedure declaration, in the same way that you would add `:far` to a declaration to force far calls.

The topics in this section are:

- [CodeWarrior Default Calling Convention](#)
- [Microsoft-Standard Conventions](#)
- [__cdecl calling convention](#)
- [__fastcall calling convention](#)

- [__stdcall Calling Convention](#)
- [The DLLEXPORT and DLLIMPORT Modifiers](#)

CodeWarrior Default Calling Convention

The CodeWarrior default calling convention has the following characteristics:

- Parameters are pushed onto the stack left to right.
- The called routine is responsible for stack maintenance.
- Names are not decorated at all.
- The case used for any symbol is the one used the first time it is declared in the calling program.

It is important to remember, when targeting Win32/x86 with CodeWarrior, that the compiler will *not* automatically assume any of the Microsoft-standard conventions for a procedure or function declared without a calling-convention qualifier.

Microsoft-Standard Conventions

The table below summarizes the three Microsoft-standard calling conventions supported in CodeWarrior Pascal.

Calling Convention	Responsible for Stack clean-up	Parameter passing
<code>__cdecl</code>	Calling routine	Parameters are pushed onto the stack right to left (in reverse order)
<code>__stdcall</code>	Called routine	Parameters are pushed onto the stack right to left (in reverse order)
<code>__fastcall</code>	Called routine	Parameters are stored in processor registers, then pushed onto the stack.

For details on each convention, see [__cdecl calling convention](#), [fastcall calling convention](#), [stdcall Calling Convention](#).

__cdecl calling convention

This calling convention is the one typically used by C and C++ programs. The calling routine is responsible for cleaning up the stack and for this reason, routines using `__cdecl` must include code to perform stack clean-up. As a result, executables exclusively or primarily using `__cdecl` compile to larger executables than executables exclusively or primarily using `stdcall`.

The table below provides implementation details for the `__cdecl` calling convention.

Implementation aspect	Implemented how?
Order in which arguments are passed	Arguments are passed right to left.
Responsibility for maintaining the stack	The calling routine must pop the arguments from the stack.
Convention respecting name-decoration	An underscore character (<code>_</code>) is prefixed to names. <code>variable</code> becomes <code>_variable</code> .
Convention respecting case translation	No case translation is performed for <code>__cdecl</code> .

fastcall calling convention

The `fastcall` calling convention specifies that whenever possible, arguments to routines will be passed in processor registers. The table below shows the implementation of this calling convention.

Implementation aspect	Implemented how?
Order in which arguments are passed	All arguments are passed right to left except for the first two arguments which are four bytes or smaller in size. These two are passed in the ECX and EDX registers.
Responsibility for maintaining the stack	The called routine must pop the arguments off of the stack.
Convention respecting name-decoration	Symbol names are prefixed by an at sign (@); for example, <code>variable</code> is prefixed to become <code>@variable</code> . They are also suffixed by an at sign, plus the number of bytes in the parameter list, expressed in decimal notation. Continuing the above example, and with a parameter list 47 bytes long, <code>@variable</code> becomes <code>@variable@47</code> .
Convention respecting case translation	No case translation is performed for <code>__fastcall</code> .
Convention respecting return values	<code>__fastcall</code> handles return values the same way <code>__cdecl</code> does; see <code>__cdecl</code> .

`__stdcall` Calling Convention

For routines using the `__stdcall` calling convention, which in a Win32 context is used to call Win32 API routines, must be prototyped. The table below shows the implementation of this calling convention.

Implementation aspect	Implemented how?
Order in which arguments are passed	Right to left
How are arguments passed? (value vs. variable)	By value, unless a pointer or reference type is passed
Responsibility for maintaining the stack	The called routine must pop the arguments off of the stack.
Convention respecting name-decoration	Symbol names are prefixed by an underscore (_); for example, <code>variable</code> is prefixed to become <code>_variable</code> . They are also suffixed by an at sign (@), plus the number of bytes in the parameter list, expressed in decimal notation. Continuing the above example, and assuming a parameter list 39 bytes long, <code>_variable</code> becomes <code>_variable@39</code> .
Convention respecting case translation	No case translation is performed for <code>__stdcall</code> .

The DLLEXPORT and DLLIMPORT Modifiers

Dynamically linked libraries (DLLs) are an accepted and widely used tool when programming for Win32. You can create Win32 DLLs, and access the routines stored in them, using CodeWarrior.

The `DLLEXPORT` modifier is critical for creating DLLs; the `DLLIMPORT` modifier is critical to accessing the routines in them. Both are used like the modifiers for the three Microsoft-standard calling-conventions:

```
program Bedrock;
```

```
var
    fred,wilma,dino,pebbles: integer; DLLIMPORT;
```

```
function trips_over_dino (fred, dino: integer):  
    boolean: DLLIMPORT;
```

with the slight difference that `DLLEXPORT` and `DLLIMPORT` can be applied not only to routines, but also to variables. Also, they can be combined with the `__stdcall` modifier.

DLLEXPORT

The `DLLEXPORT` modifier works this way:

- Create a project with **Win32 x86 Linker** selected in the **Target Settings** panel, and choose **Dynamic Link Library** for **Project Type** on the **x86 Target** panel.
- In your source code of your DLL project, apply the `DLLEXPORT` modifier to the routines and variables that you want to make externally accessible.
- After compilation, ensure that your DLL is someplace the operating system will automatically look for it. Typically this can be any of:
 - The directory into which your executable has been installed
 - The Win32 operating system main directory (`C:\WINDOWS`, `C:\WINNT`, or equivalent)
 - The Win32 operating system `SYSTEM` subdirectory (`C:\WINDOWS\SYSTEM`, `C:\WINNT\SYSTEM`, or equivalent)

DLLIMPORT

The `DLLIMPORT` modifier works this way:

- Create a Pascal source file which is not the main one for your project. The implementation part of this extra file should be empty; the interface section contains only declarations for the exported DLL routines and variables you want to be accessible to your project.

Pascal for Win32/x86

Calling Conventions for Win32/x86

NOTE: It is not necessary, in this file, to reference in any way the DLL containing the exported routines and/or variables.

- In the `USES` clause of the main source file for your project, reference the extra Pascal source file you created above, as a unit.
- Reference the exported DLL variables and/or call the exported DLL routines in the main source file for your project.



Inline x86 Assembler

This chapter describes support for inline assembly language programming built into the CodeWarrior compilers.

This chapter does *not* discuss the stand-alone assembler available for x86. For information on the stand-alone assembler, see the *Assembler Guide*.

This chapter does not document all the instructions available in x86 assembly language. For information on x86 assembly language instructions, see *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual* by Intel Corporation.

For information on Advanced Micro Devices, Inc. (AMD™) 3DNow!™ Technology, see *3DNow!™ Technology Manual*, published by AMD™ (publication # 21928).

The sections in this chapter are:

- [Working With Inline Assembly for x86](#)

Working With Inline Assembly for x86

This section describes how to use the CodeWarrior compiler's built-in support for assembly language programming, including assembler syntax.

The topics in this section are:

- [Inline Assembler Syntax for x86](#)
- [Using Local Variables and Arguments in x86 Assembly](#)

Inline Assembler Syntax for x86

To specify that a block of code in your file should be interpreted as assembly language, use the `asm` keyword.

To ensure that the C/C++ compiler recognizes the `asm` keyword, do not enable the **ANSI Keywords Only** option in the **C/C++ Language panel**. This panel and its options are fully described in the *C Compilers Reference*.

The assembly instructions are the standard x86 instruction mnemonics. For information on x86 assembly language instructions, see the *Assembler Guide*.

The x86 inline assembler supports `asm` blocks within a function. Inline assembly code for x86 uses the following syntax:

```
asm (single instruction )
```

or

```
asm { multiple instructions }
```

or

```
__asm single instruction
```

Nothing else can follow the statement on the same line except a comment or another `__asm` statement, for example:

```
__asm statement __asm statement
```

This form is useful for defining macros that have inline assembly.

NOTE: Variables references by inline assembly instructions will be forced into memory and cannot be allocated to processor registers by the optimizer.

An `asm` instruction or block may be used wherever a statement is allowed.

NOTE: Functions that contain an `asm` block are never optimized, regardless of compiler settings.

Supported Directives in Inline Assembly

CodeWarrior supports the **ALIGN**, **DB**, **DW**, **DD**, **EMIT** directives in inline assembly:

ALIGN	align next instruction to next DWORD
ALIGN n	align next instruction to n-byte boundry
ALIGN n,m	align next instruction to n-byte boundry, if such boundry is within m bytes
DB [byte-value]*	directly insert bytes into instruction stream
DD [word-value]*	directly insert words (2-bytes) into instruction stream
DW [dword-value]*	directly insert dwords (4-bytes) into instruction stream
EMIT byte	directly insert single byte into instruction stream

Labels Supported

Labels are also supported in inline assembly. They have the same syntax as C labels, for example:

```
__asm {  
    MOV EAX, 8  
loop_top:  
    DEC EAX  
    JNZ loop_top  
}
```

Using Local Variables and Arguments in x86 Assembly

Assembly instructions may refer to local and global variables as operands. They can use the name of a structure, class, or union as an immediate operand that evaluates to the size of the structure. To specify the offset of a member, use the structure, union, or class name qualified with the member name, separated by a dot (.).



Debugging for Win32/x86

This chapter covers only those aspects of debugging that are specific to the Win32/x86 platform; see the *Debugger User Guide* for basic information on the debugger in general.

This chapter includes the following topics:

- [Project Settings for Debugging](#)
- [Debugging an Application](#)
- [Debugging a DLL](#)
- [Debugger Preference Panels](#)
- [Setting Up for Remote Debugging](#)

Project Settings for Debugging

The CodeWarrior debugger requires symbolic information in order to operate. The CodeWarrior back-end compiler for x86 processors can generate this information in either of two formats: SYM or CodeView. The choice is determined via project settings in the **x86 Processor** and **x86 Linker** settings panels; see [“x86 Processor” on page 87](#) and [“x86 Linker” on page 92](#) for more information. SYM information is stored in a separate file called a SYM file; CodeView information is stored in the object file itself, whether an executable application or a DLL.

Typically, you'll want to generate symbolic information for all source files in a project, so that the debugger can display high-level source code as you debug. In some cases, however, you may want to disable the generation of debugging symbolics for certain files. You can turn debugging on or off for a file by toggling the check

mark next to its name in the **Project** window; see the *Debugger User Guide* for details.

For more distinct control over which functions within a source file generate debugging information, you can use the C/C++ **sym** pragma. Both the **Generate SYM File** option in the **x86 Linker** project settings panel and the **SYM Format** option in the **x86 Processor** panel must be turned on. You can then use the lines

```
#pragma sym on  
and  
#pragma sym off
```

in your source code to control the generation of symbolic debugging information for specific functions. See the *C Compilers Reference* for more information.

Debugging an Application

When you double-click a program's symbol file, the IDE will launch and automatically run the program in Debug mode. If your main entry point is in a module that has debugging information enabled, the debugger will stop program execution at the `main( )` function so you can set breakpoints and other debugging information. If the main entry point is not in a module with debugging information, go to the **Global Preferences** panel and deselect the option **Automatically launch applications when SYM file opened**. This gives you a chance to set breakpoints before actually letting the program run.

See also *Debugger User Guide* for more information on preferences for the debugger.

Debugging a DLL

When your program includes dynamic link libraries (DLLs), you can also debug those. To debug a DLL, open its symbol file before launching the application. You can set breakpoints in the DLL, and the debugger will automatically honor them when the application

that uses the DLL is run. Before starting the program, make sure the DLL is in the same folder with it.

NOTE: The Mac-OS–hosted debugger does not automatically transfer DLLs to your Win32/x86 system. Make sure the DLL is installed there before starting the debugging process.

Debugger Preference Panels

The debugger is integrated seamlessly with the rest of the CodeWarrior integrated development environment.

This section discusses one specific debugger preference panel:

- [Runtime Settings](#)

Runtime Settings

The **Runtime Settings** panel is shown in [Figure 12.1](#). This panel consists of two main areas: **Environment Settings** and **General Settings**.

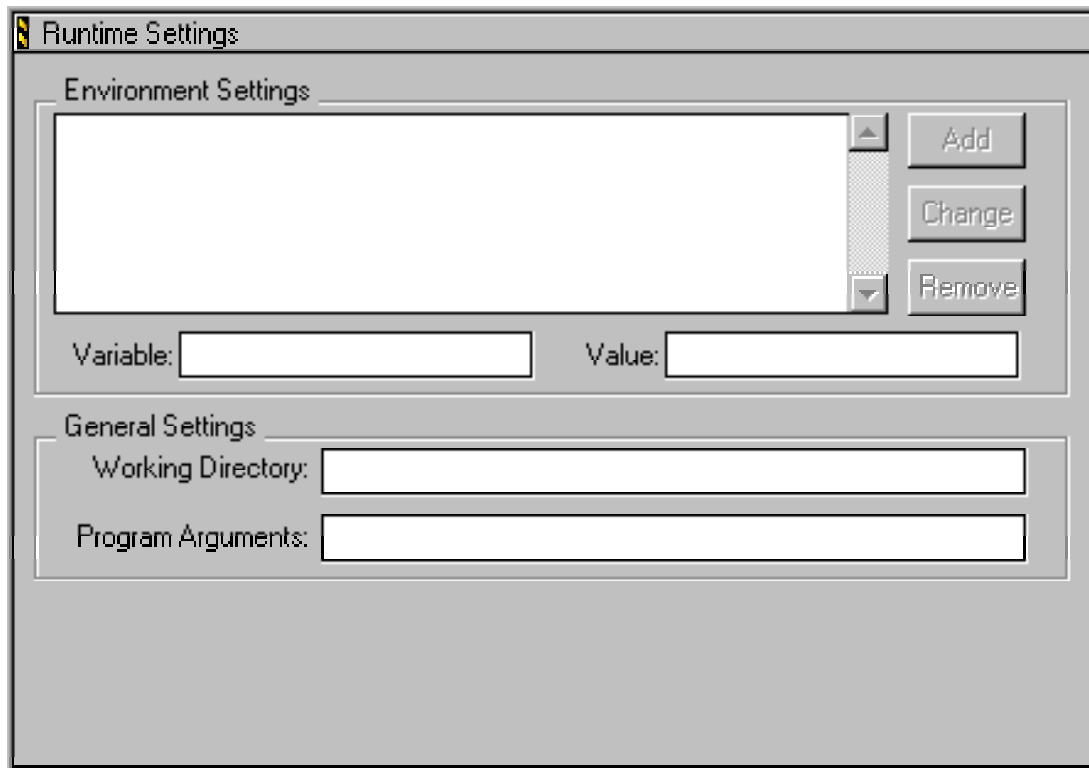
The **Environment Settings** area allows you to specify environment variables that are set and passed to your program as part of the `envp` parameter in the `main()` or available from the `getenv()` call and are only available to the target program. When the your program terminates, the settings are no longer available. For example, if you are writing a program that logs into a server, you could use variables for `userid` and `password`.

The **General Settings** area has the following fields:

- **Working Directory** — specifies the directory in which debugging occurs. If no directory is specified, debugging occurs in the same directory the executable is located.
- **Program Arguments** — passes command-line arguments to your program at start-up. Your program receives these arguments when started with the **Run** command.

For additional information about the **Runtime Settings** panel, see [“Runtime Settings” on page 95](#).

Figure 12.1 Runtime Settings



Setting Up for Remote Debugging

The Mac OS–hosted CodeWarrior tools include remote debugging capabilities for Win32/x86 applications. This is a two-machine debugger setup that runs on the Mac OS host and communicates with the running application via a remote communications nub on the x86 target computer, using either a serial connection or a TCP/IP network connection. This section covers only those aspects of debugging that are specific to the Win32/x86 platform and the remote debugger; see the *Debugger User Guide* for basic information on the debugger in general.

To debug Win32/x86 programs with the Mac-OS-hosted debugger, you must

- set up a physical connection between your Mac OS host machine and your Win32/x86 target system, and
- install and configure the connection software for communicating between the two systems.

The following sections tell how to accomplish these tasks:

- [Connecting the Computers](#)
- [Configuring the Connection Software](#)

Connecting the Computers

There are two ways of connecting your Mac OS and Win32/x86 computers:

- Serial connection
- Network connection

Which method you choose will depend on your physical situation and whether you are connected to a local-area network.

Serial connection

The simplest way to connect your host and target computers is via a serial communications line. Get a five-wire null modem cable with a Macintosh serial port connector on one end and a PC serial connector on the other, or use suitable adapters. Then just plug the two computers together, from one serial port to the other.

Network connection

If both of your computers are connected to a local-area network, you can use the TCP/IP protocol to communicate between them. If you are part of a network that uses TCP/IP, then you're already set up; all you need to make the connection is the TCP/IP address of your Win32/x86 computer. If you aren't already networked, set up both your Mac OS system and your Win32/x86 system to communicate via TCP/IP as follows:

- On the Mac OS system, use the **TCP/IP** control panel to enable and set up TCP/IP. If your network is not already using TCP/IP, you can choose any address (for example, 1.1.1.1). If your network is already set up for TCP/IP, you must get a unique address from your network administrator.
- On the Win32/x86 side, you can add the TCP/IP protocol and configure it from the **Network** control panel. Check **Installed Network Software** and look for **TCP/IP Protocol**. If TCP/IP is installed, double-click it to configure or to get the address, which you must have to set up communications. If TCP/IP is not installed, install it by clicking the **Add** button. Again, if you were not already using TCP/IP software on your network, you can choose any address different from the one you used on the Mac OS side (for instance, 1.1.1.2).

Configuring the Connection Software

Establishing a communications link between your host and target computers requires some low-level connection software on both the Mac OS and Win32/x86 sides. You must install the needed software and then configure it appropriately for the type of connection you're using.

Configuring the host system

In order to work for remote debugging, the debugger on your Mac OS host system requires a few system extensions to be installed in the **Extensions** subfolder of your system folder. The CodeWarrior Installer program will automatically place these items in your **Extensions** folder at installation time. Then, before debugging a Win32/x86 project, you must configure the debugger on the Mac OS host for remote operation. Run the IDE and open the project file you wish to debug on the Win32/x86 target:

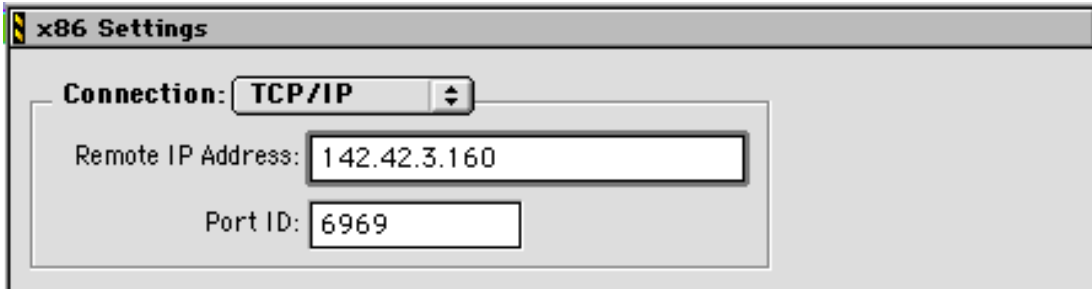
1. **Display the Preferences window.**

Choose **Preferences** from the **Edit** menu. The **IDE Preferences** dialog box appears.

2. **Select the x86 Settings preference panel from the list of preference panels on the left side of the IDE Preferences dialog box.**

The **x86 Settings** panel appears as in [Figure 12.2](#).

Figure 12.2 x86 Settings panel for the debugger



3. Set the connection mode.

On the pop-up menu labeled **Connection**, select either

- **Modem Port** or **Printer Port** for a serial connection, or
- **TCP/IP** for a network connection

4. Complete the Remote IP Address field in the x86 Settings preference panel, as shown in [Figure 12.2](#).

This should be the IP address of your Mac OS computer. You can find the IP address of your Mac OS computer in the MacTCP or TCP/IP Control Panels. If you can't find it there, ask your system administrator.

NOTE: Do not change the **Port ID** field in the **x86 Settings** panel. This number is fixed.

5. Confirm your settings.

Click the **Save** button to accept your preference settings and close the dialog.

Debugging for Win32/x86

Setting Up for Remote Debugging



Libraries and Runtime Code for Win32/x86

This chapter describes the Win32/x86-specific features of standard libraries provided with CodeWarrior.

Standard Libraries for Win32/x86 Overview

A variety of static libraries for use with the CodeWarrior development environment exist. They include ANSI-standard libraries for C and C++, as well as libraries for software development with the Win32 application programming interface and the Microsoft Foundation Classes. This chapter discusses how to use these libraries when targeting the Win32/x86 platform. For more on the common (target-neutral) features of the standard libraries, see the *MSL C Reference* and the *MSL C++ Reference*.

Topics in this chapter are:

- [The Metrowerks Standard Libraries](#)
- [Libraries for Win32 Development](#)
- [The Microsoft Foundation Classes](#)
- [Win32-Specific Functions in MSL C](#)

The Metrowerks Standard Libraries

The Metrowerks Standard Libraries (MSL) are ANSI-standard C and C++ libraries included with the CodeWarrior product:

Libraries and Runtime Code for Win32/x86

The Metrowerks Standard Libraries

`Ansicx86.lib` ANSI-standard C

`ANSICPPX86.LIB` ANSI-standard C++

These libraries are fully documented in the *MSL C Reference* and *MSL C++ Reference*. In addition, there is a Win32/x86 run-time library that contains startup code required to initialize applications and dynamic link libraries (DLLs):

<code>Mwcrtl.lib</code>	Win32/x86 run-time support
<code>Mwcrtd.lib</code>	Debug version of runtime library
<code>Mwcrtlmse.lib</code>	Runtime supporting Microsoft-compatible C++ exceptions
<code>Mwcrtdmse.lib</code>	Debug runtime supporting Microsoft-compatible C++ exceptions
<code>Mwcrtl40.dll</code>	DLL version of standard runtime
<code>Mwcrtdll.lib</code>	Import library for above DLL
<code>Mwcrtd40.dll</code>	DLL version of debug runtime
<code>Mwcrtdlld.lib</code>	Import library for above DLL
<code>Mwcrtlmse40.dll</code>	DLL version of runtime w/ MS Exception support
<code>Mwcrtdllmse.lib</code>	Import library for above DLL
<code>Mwcrtdmse40.dll</code>	DLL version of debug runtime w/ MS Exception support
<code>Mwcrtdlldmse.lib</code>	Import library for above DLL

NOTE: The DLL versions of the runtime library also include the MSL C library, so if you use those, you do not also need to include `ANSICx86.LIB` in your project.

All of these libraries are provided in both static-library and source-code formats for inclusion in your project. To use them, simply add them to your project with the **Add Files** command from the **Project** menu in the IDE.

NOTE: The C++ library depends on the presence of the C library, so if you use it you must include the C library as well. All Win32/x86 applications and DLLs should also include the runtime library.

File and Memory Restrictions

If your final build consists of separate elements, such as an application and a DLL, and you include the MSL libraries in both, there can be complications. If you create both an application and a DLL with the same MSL library, you will end up with two copies of the library in the final code: one in the application and one in the DLL. Files and memory opened or allocated from one cannot be used indiscriminately by the other.

For example, you cannot open a file in one application program or DLL and then use it in another one. All operations on a given file must be performed within the same application or DLL. This is because the two copies of the library each have their own copy of the internal structures associated with a file.

For memory, the restriction is less severe but even more important. Memory allocated by one application or DLL can be used by another, but *it must not be freed except in the same application or DLL that allocated it*. If you break this rule, you will probably trash the heap and eventually crash.

Exception Handling in MSL C++

A related issue concerns C++ exception handling. When you use exception handling, the compiler and linker build tables that direct the action when an exception is thrown. This is no problem if all your code is in one application or DLL. A problem does arise, however, if you throw an exception in one DLL and catch it in another, or catch it in the application, because the thrower's and catcher's copies of the MSL library each have only half the needed information.

To avoid this difficulty, the C++ exception-handling code is provided in the form of a separate DLL named `CPPEXC.DLL`. It is

strongly recommended that you use this DLL for C++ exception handling. This ensures that there is one shared copy of the exception-handling code, so that when you throw an exception, the code is aware of both the thrower and catcher, even if they are in separate applications or DLLs.

To use the C++ exception DLL, include the library

`CPPEXC.DLL.LIB`

in your project. This library must precede the standard run-time library, `Mwcrtl.lib`, in your project. The linker will then link with the exception-handling code in the corresponding DLL

`CPPEXC.DLL`

rather than with that in the standard run-time library. This DLL must then be present on the target Win32/x86 system when your application is run, and must be in the same directory as the application or in any directory mentioned in the `PATH` environment variable.

Libraries for Win32 Development

In addition to the MSL libraries, CodeWarrior also comes with a set of standard Microsoft libraries embodying the Win32 application program interface (API). The following libraries are required for most Win32 applications:

<code>KERNEL32.LIB</code>	Win32 kernel
<code>GDI32.LIB</code>	Win32 graphics
<code>USER32.LIB</code>	Win32 graphical user interface (GUI)

Depending on what specific Win32 facilities you use, you may also need to include one or more of the following libraries:

<code>ADVAPI32.LIB</code>	<code>OLEAUT32.LIB</code>	<code>THUNK32.LIB</code>
<code>CAP.LIB</code>	<code>OLEDLG.LIB</code>	<code>UUID.LIB</code>
<code>COMCTL32.LIB</code>	<code>PENTER.LIB</code>	<code>VERSION.LIB</code>

COMDLG32.LIB	RASAPI32.LIB	VFW32.LIB
LZ32.LIB	RICHED32.LIB	WINMM.LIB
MPR.LIB	RPCNDR.LIB	WINSPOOL.LIB
MSACM32.LIB	RPCNS4.LIB	WINSTRM.LIB
OLE32.LIB	RPCRT4.LIB	WSOCK32.LIB
	SHELL32.LIB	

The Microsoft Foundation Classes

The Microsoft Foundation Classes (MFC) are an object-oriented application framework that implements the key building blocks in a Win32 application. To use the MFC library, start your project from MFC-based stationery. This stationery includes both the static MFC library (MFC.LIB) in the project and the necessary precompiled header file (MFCHeaders) in the project prefix. You can also add the library manually if you wish.

Win32-Specific Functions in MSL C

A number of functions have been added to the Metrowerks Standard C Library for the Win32 target. These functions are mostly in support of the Microsoft Foundation Class (MFC) library. This section discusses details on each of these additional functions, which you are free to use directly yourself should the need arise. The functions are:

_access	_gcvt	_strdate
_alloca	_getcwd	_strdup
_chdir	_heapmin	_stricmp
_chdrive	_isatty	_strnicmp
_fdopen	_itoa	_strrev
_fileno	_makepath	_strupr

Libraries and Runtime Code for Win32/x86

Win32-Specific Functions in MSL C

<code>_searchenv</code>	<code>_fullpath</code>	<code>_get_osfhandle</code>
<code>_getdcwd</code>	<code>_getdiskfree</code>	<code>_getdrive</code>
<code>_getdrives</code>	<code>_itow</code>	<code>_open_osfhandle</code>
<code>_splitpath</code>	<code>_ultoa</code>	<code>_wcsdup</code>
<code>_wcsicmp</code>	<code>_wcslwr</code>	<code>_wcsnicmp</code>
<code>_wcsnset</code>	<code>_wcsrev</code>	<code>_wcsset</code>
<code>_wcsupr</code>	<code>_wstrrev</code>	<code>_wtoi</code>

For more information, see the *Microsoft Win32 Reference* and *MSL C Reference*, included on the CodeWarrior CD.



Optimizing Code for AMD-K6®

This chapter describes the optimizations that can be performed when targeting the AMD-K6® processor.

Optimizing for AMD-K6® Overview

CodeWarrior is capable of generating code that takes advantage of speed optimizations available with the AMD-K6® processor.

Topics in this chapter are:

- [3DNow!™ Optimizations](#)
- [Pragma Support](#)
- [Library Considerations](#)

3DNow!™ Optimizations

The topics in this section are:

- [Calling Conventions](#)
- [Minimizing Mode Switching](#)
- [Vectorization](#)

Calling Conventions

When writing code for the AMD-K6® processor, you should be aware that the Win32 `__stdcall` calling conventions exact a performance penalty on the execution of AMD-K6® or MMX code.

If you enable the use of special calling conventions, the compiler will generate code that preserves fewer registers in the AMD-K6® processor, enabling faster execution. To enable these calling conventions, select the **Use MMX/3DNow! Calling Convention** checkbox in the [x86 Processor](#) panel. Refer to [“x86 Processor” on page 87](#) to learn how to do this.

When using 3DNow!™ calling conventions, the whole application program (all source files and library source files) should be compiled using the 3DNow!™ calling conventions. Any inconsistency caused by intermixing calling conventions in source files, library code and assembler code can lead to runtime errors.

WARNING! The `UseMMX/3DNow!CallingConvention` option is inappropriate for functions that are exported as external entry points, and, thus, can be called from other binaries that are not aware of 3DNow! calls. Returning from MMX or 3DNow! code to x86 code in another module without having done an FEMMS is very bad news.

Minimizing Mode Switching

The key to achieving faster runtime performance with the AMD-K6® is minimizing mode switching. For this reason, **3DNow!** mode performs better with code that contains `int` and `float` types. If your code contains math that uses `float` and `double` types, you will incur more mode switching, which will be expensive at runtime because of the AMD-K6® mode switch operation.

Since the C and C++ languages assume that floating-point constants are doubles unless explicitly labeled otherwise, you need to be careful how you write your code. If you intend to write your code to keep only `int` and `float` types, but you want to avoid `double` variable types, you need to do some special things. For example, if you need to define a floating-point constant called `MyFudgeFactor` having a value of 2.0 as a `float` type, you would write:

```
#define MyFudgeFactor 2.0f
```

instead of:

```
#define MyFudgeFactor 2.0
```

2.0f needs to be specified because otherwise the compiler assumes that you want a type `double` for the constant.

Vectorization

The CodeWarrior compilers perform a limited amount of vectorization of the optimized code.

In order to get this performance benefit, the code has to be written in a certain way. See [Loop Vectorization](#) for more information on writing code that takes advantage of vectorization.

NOTE: Some of these restrictions may be relaxed in a future release of the tools.

The `restricted` keyword is a C++ language feature that may be implemented as some future date, since it's in the C++ draft standard.

Loop Vectorization

Loop vectorization is a transformation that enables the compiler to generate vector instructions for a processor that supports these instructions, such as a Pentium with MMX and AMD 3DNow!. The optimizer looks at operations and operands in loops to decide whether vector instructions can be used to combine operations and operands. This results in a larger number of operations per cycle and less loop branching per iteration.

Loop vectorization will be utilized if the code is a counted loop that goes up, only contains assignment statements, and has no control code or pointer dereferences. This is useful for doing operations on static or local arrays (arrays with a known base value).

Loop Example

The following example shows the loop form that the optimizer can optimize for AMD™ targets:

```
for( i = initial; i op N; i = i + 1 ) {  
    ... = a[ i + k1 ];  
    a[ i + k ] = ...;  
    ... = a[ i + k2 ];  
    a[i+k3] = ...  
  
}
```

Where “i” is the loop index and “a” is a static array. The references on the right side of the assignment represent loads of “a” and the references on the left represent stores into “a”. The expression on the right side of the assignment can, in general, have more than one vector operand, constants, loop invariant scalars or expressions and vector operations.

Loop Restrictions

The following restrictions should be observed when writing loop structures:

- op implies < or <= operator
- i has a unit stride
- i is only used in a unit stride dimension of an array access
- body of the loop only uses i in index expressions
- i is the inner loop index of a loop nest
- only inner loops are considered for vectorization
- no control flow in the loop
- no calls in the loop

Dataflow Restrictions

The following dataflow restrictions must also be observed:

- Loop carried data dependencies are not allowed

In the Vectorization Loop example, the loop could not be vectorized if any of the following conditions existed:

$k > k1$

or

$k2 > k$

or

$k3 > k$

For example, the following loop cannot be vectorized because doing so would change its semantics ($k > k1$)

```
for( i = initial; i op N; i = i + 1 ) {
```

```
    b[i]= a[ i - 1 ];
```

```
    a[ i ] =b[i] ;
```

```
}
```

Similarly, the following loop can also not be vectorized as doing so would change its semantics ($k2 > k$)

```
for( i = initial; i op N; i = i + 1 ) {
```

```
    a[ i ] = b[i];
```

```
    b[i]= a[i+1];
```

```
}
```

Similarly, the following loop can also not be vectorized as doing so would change its semantics($k3 > k$)

```
for( i = initial; i op N; i = i + 1 ) {  
  
    a[ i ] = b[i];  
  
    a[i+1] = b[i];  
  
}
```

All loads and stores must be disambiguated at compile time

Vectorization requires that all loads and stores be disambiguated at compile time. If there is a load or store through a pointer in the loop, vectorization will not be possible.

Pragma Support

There are three pragmas that you can use to enable the various MMX and AMD-K6® code generation options. The #pragma statements are:

- #pragma mmx—corresponds to the [MMX](#) checkbox
- #pragma k63d—corresponds to the [3DNow!](#) checkbox
- #pragma k63d_calls—corresponds to the [Use MMX/3DNow! Calling Conventions](#) checkbox

Refer to [“x86 Processor” on page 87](#) for a description of each of these settings. To learn more about #pragma directives, refer to the *C Compiler Reference*.

NOTE: The k63d #pragma is not Pentium-compatible.

NOTE: The __stdcall keyword overrides the #pragma.

Library Considerations

To get maximum performance benefits from 3DNow!™ you would need float-based libraries, as double-based libs will slow down performance due to mode switching. Refer to [“Minimizing Mode Switching” on page 142](#) for more information on mode switching and mixing of types.

WARNING! The libraries supplied with CodeWarrior are double-based libraries. You will want to build your own libraries based on float types to get maximum performance.

If you prototype the `fsqrt()` and `fabs()` functions using float parameters in your code, the code will inline to a single K63D instruction. They will be much faster than the CodeWarrior-supplied library functions since they don't cause a mode switch.

NOTE: You must turn on **InlineIntrinsicFunctions** in the [x86 Processor](#) settings panel to get this single-instruction inlining to work. To learn how to do this, refer to [“x86 Processor” on page 87](#).

You have to manually create these prototypes that use float in your own header file, these prototypes aren't in the CodeWarrior-supplied headers.

Figure 14.1 Examples of prototypes that will generate single 3DNow! instructions

```
float fabs(float x); /* ansi is double fabs(double x); */  
float sqrt(float x); /* ansi is double sqrt(double x); */
```



Troubleshooting

This chapter covers some frequently asked questions (and answers) about CodeWarrior tools on Windows 95 or Windows NT 4.0. If you have a problem, come here first. Others may have encountered similar difficulties, and there may be a simple solution.

Troubleshooting Overview

This chapter discusses various problems people have encountered while writing and debugging their code. There are suggested solutions for each problem. If your problem is not in this chapter, please contact Metrowerks Technical Support for assistance. This chapter will expand to include more problems with each release of CodeWarrior.

NOTE: Currently, this section is small, but will expand in the future as more problems are encountered. If you have a problem that isn't addressed here, send a note to Metrowerks Technical Support. See the front of this manual for contact information.

The topics covered include:

- [Win32 Programming Problems](#)

Win32 Programming Problems

This section covers known problems that relate to programming in the Win32 environment. Not all problems are specific to CodeWarrior.

There is one primary topic in this section:

- [Errors With min\(\) and max\(\)](#)

Errors With `min()` and `max()`

Problem

I'm trying to port C/C++ code written with a different programming environment. I'm encountering the following error message from `mlimits.h`:

```
Error : ', ' expected
mlimits.h line 70 static T min() throw();
```

Solution

This is actually a known issue with Win32 programming, it's not specific to CodeWarrior. The Win32 API defines two macros named `min()` and `max()` in `windef.h`. These conflict with the ANSI C++ `min()` and `max()` template functions, however. A workaround is to `#define` a symbol called `NOMINMAX` in a prefix file, or before you `#include <windows.h>`. This define shuts off the Win32 `min()` and `max()` macros, preventing the conflict with the ANSI versions.

See also The Microsoft Win32 SDK documentation.

Compiling a Project for Mac OS on Windows NT

The Scenario

You've successfully made a Windows executable and would like to compile it for the Macintosh on Windows NT. However, when you try to compile your program, it seems to do nothing. The executable appears not to exist, yet you don't receive an error message. What is a programmer to do?

NOTE: You can compile a Windows executable on a Macintosh only if your code uses the C and C++ standard libraries. If your project uses operating-system specific code, you have to maintain separate code bases for each platform.

Behind The Scenes

Actually, your program's executable is there: It exists as a 0-byte file with the same name as your project. CodeWarrior's Windows-hosted Macintosh tools deal with the data/resource fork situation by emitting two files: a data fork image file and a resource fork image file. Because they have the same name, the resource fork image file is put into a folder called `Resource.frk`. You must copy both the 0-byte data fork image and the `Resource.frk` folder onto a FAT formatted floppy disk before bringing the executable over to the Macintosh using PC Exchange (which will merge the two image files back together).

NOTE: After you copy the files over to the Macintosh, remember to use ResEdit to set the creator and type of the binary to the application so the Macintosh will recognize it as an executable.

An Alternative Solution

An easier way to compile Macintosh programs created on Windows NT exists. Follow these steps using CodeWarrior target settings panels:

1. **Display the Target Settings dialog box for the project you created on Windows NT.**

Choose the *Target Settings* option from the **Edit** menu in the IDE, on which *Target* is the target of your current project.

2. **Select the Target Settings panel from the list of settings panel on the left side of the Target Settings dialog box.**
3. **Choose MacBinarize Post Linker from the post-linker pop-up menu.**

This will create the executable packaged as a MacsBin archive.

4. **Copy this file over to your Macintosh and unpack it with Stuffit Expander, or another decompression program.**

When it is unstuffed, it should be a full-fledged Macintosh executable with the correct creator and type.

Troubleshooting

Win32 Programming Problems

Index

Symbols

`__cdecl` 118
`__fastcall` 118
`__stdcall` 119
`__stdcall` conventions 141

Numerics

3DNow! Technology 11

A

Access Paths panel 78
Add Files command 136
Add option, Debugger Runtime Settings panel 96
alignment boundary 89
allocating variables for x86 106
AMD 11
 3DNow! 11
AMD K6 141–??, 147, ??–147
 libraries 147
 mode switch 142
ANSICPPX86.LIB library 136
Ansicx86.lib library 136
ANSI-standard libraries 135
application files. *See* executable files
applications
 console 94
 creating 51–61
 debugging 128
 defined 51
 entry-point function 94
 initialization 136
 version number 95
 Windows GUI 94
asm blocks in a function 124
asm keyword 124
Assembler 126
assembler
 See also inline assembler
Automatically launch applications
 when SYM file opened preference
 setting 128

B

back-end compiler *See* compiler
Base Address option, x86 Target panel 86
bool size 103
Borland International, Inc. 25
BoundsChecker debugger (NuMega) 25
building projects 60
Byte Alignment option, x86 CodeGen panel 89

C

C Compilers Reference 14, 59, 60, 64, 66, 68, 69, 71, 74, 128
C/C++ Language panel 59
C/C++ names 67, 68, 69, 70
C/C++ programming language
 data structures 24
 multiple inheritance 25
 single inheritance 24
C/C++ run-time library. *See* libraries, C/C++ run-time
C/C++ compiler
 described 21
Calling conventions 116
 `__cdecl` 118
 `__fastcall` 118
 `__stdcall` 119
calling conventions for x86 104–106, 116–122
calling conventions, functions 24
Change option, Debugger Runtime Settings
 panel 97
char size 103
CodeView debugging information 24, 95, 127
CodeView Format option, x86 CodeGen panel 90
CodeWarrior
 C/C++ compiler described 21
 compare to command line 25
 compatibility with Microsoft software 24–25, 77–78
 compiler architecture 18
 components 19–24
 Constructor resource editor 76
 debugger described 20
 debugging 61
 described 11

Index

- development process 25–28
 - documentation architecture 12
 - Documentation folder 15
 - IDE debugger 129
 - IDE described 19
 - Installer 132
 - installing 18
 - integrated development environment (IDE) 129
 - linker described 22
 - Pascal compiler described 21
 - projects, building 60
 - projects, creating 51–54
 - tools listed 19
 - Tutorial 29–50
 - tutorials 15
 - Win32 resource compiler (RC) 22, 75, 76–78
 - x86 back-end compiler 22, 24, 127
 - x86 linker 22–23, 24
 - CodeWarrior Documentation folder 15
 - COFF. *See* Common Object File Format
 - Command File 67
 - command file. *See* linker command file
 - Command File option, x86 Linker panel 95
 - command line (DOS) 51
 - command line arguments
 - new 11
 - command-line and CodeWarrior compared 25
 - commands
 - linker. *See* linker commands
 - menu. *See* menu commands
 - Committed Heap Size option, x86 Target panel 86
 - Committed Stack Size option, x86 Target panel 86
 - Common Object File Format (COFF) 22–23, 25
 - dynamic link libraries (DLLs) 23
 - executable files 23, 24
 - library files 23
 - object modules 22, 23, 24
 - communications nub 21, 130
 - compatibility with Microsoft software 24–25, 77–78
 - compiler
 - architecture 18
 - back-end for x86 101–109, 111–122
 - other documentation 101, 111
 - support for inline assembly 123
 - vectorization 143
 - See also* C Compilers Reference
 - See also* Pascal Compilers Reference
 - compilers
 - Win32 resource (RC) 22, 75, 76–78
 - x86 back-end 22, 24, 127
 - compiling 27, 60
 - connecting computers for remote debugging 131–133
 - configuring host (MacOS) system 132–133
 - configuring target (Win32/x86) system 130–133
 - network (TCP/IP) connection 131–132
 - serial connection 131
 - connection mode 133
 - Connection** preference setting (debugger) 133
 - control panels
 - Network** (Windows) 132
 - TCP/IP** (Mac OS) 132
 - CPPEXC.DLL library 137
 - creating
 - applications 51–61
 - dynamic link libraries (DLLs) 65–74
 - projects 51–54
 - resources 75–78
 - static libraries 63–64
 - Cross Platform Debugging 130–133
- ## D
- data structures
 - C/C++ programming language 24
 - Data types
 - DOUBLE for x86 116
 - INTEGER for x86 113
 - LONGREAL for x86 116
 - REAL for x86 115
 - SINGLE for x86 116
 - UNSIGNEDBYTE 114
 - UNSIGNEDBYTE for x86 114
 - UNSIGNEDLONG for x86 115
 - UNSIGNEDWORD for x86 115
 - deadstripping unused code 109
 - Debugger
 - described 20
 - debugger

-
- See also Debugger User Guide*
- Debugger Preferences
 - Global Preferences Panel 128
 - Runtime Settings 129
 - Debugger Runtime Settings panel 95–97
 - Debugger Settings panel 97–98
 - debugger symbol (SYM) file 61, 95, 127, 128
 - Debugger User Guide* 14, 61, 127, 128, 130
 - debuggers
 - BoundsChecker (NuMega) 25
 - Developer's Studio (Microsoft Visual C++) 25
 - SoftICE (NuMega) 25
 - debugging 28, 61
 - applications 128
 - dynamic link libraries (DLLs) 128
 - project settings 127–128
 - remote. *See* remote debugging
 - debugging information 128
 - CodeView 24, 95, 127
 - symbol (SYM) file 61, 95, 127, 128
 - declaration specifiers
 - `__declspec(dllexport)` 66, 67, 70
 - `__declspec(dllimport)` 68, 71
 - `__stdcall` 69, 70, 71
 - `__declspec(dllexport)` declaration specifier 66, 67, 70
 - `__declspec(dllimport)` declaration specifier 68, 71
 - decorated names. *See* names, decorated
 - development tools 19–24
 - CodeWarrior Constructor resource editor 76
 - CodeWarrior IDE (integrated development environment) 129
 - IDE debugger 129
 - MFC library 135, 139
 - Microsoft Foundation Classes (MFC) application framework 15, 24
 - Microsoft resource compiler 77–78
 - Win32 Application Programming Interface (API) header files 23
 - Win32 resource compiler (RC) 22, 75, 76–78
 - x86 back-end compiler 22, 24, 127
 - x86 linker 22–23, 24
 - dialogs
 - Name new project** 53, 54
 - directives supported 125
 - disassembler 92
 - DLL 121
 - .dll file type. *See* file types, .dll
 - DLLEXPORT modifier
 - overview 120
 - using 121
 - DLLIMPORT modifier
 - overview 120
 - using 121
 - _DllMain hook function 73
 - _DllMainCRTStartup entry-point function 73, 94
 - DLLs 120
 - DLLs. *See* dynamic link libraries
 - documentation
 - C Compilers Reference* 14, 59, 60, 64, 66, 68, 69, 71, 74, 128
 - CodeWarrior Documentation folder 15
 - Debugger User Guide* 14, 61, 127, 128, 130
 - IDE User Guide* 14, 51, 60, 61, 63, 64, 65, 74, 78, 92
 - Microsoft Win32 Reference* 76, 140
 - MSL C Reference* 14, 135, 136
 - MSL C++ Reference* 14, 135, 136
 - Pascal Compilers Reference* 14, 59, 60, 64, 74
 - Pascal Language Reference* 14
 - documentation architecture 12
 - DOS command line 51
 - DOUBLE 116
 - double size 104
 - Dynamic link libraries 120
 - dynamic link libraries (DLLs) 23, 24, 25, 63, 85
 - creating 65–74
 - debugging 128
 - defined 65
 - entry-point function 71–72, 94
 - exported names 66–71
 - exporting names 66–67
 - hook function 73
 - importing names 68
 - initialization 71–73, 136
 - linker files 66
 - name transformations 68–71
 - Dynamic link library 121
-

Index

E

Edit menu
 Preferences command 132
Edit menu (CodeWarrior IDE)
 Settings command 55
editing code 27
 See also IDE User Guide
Entry Point Usage option, x86 Linker panel 93
Entry Point Usage setting 72, 74
entry-point functions
 console applications 94
 dynamic link libraries (DLLs) 71–72, 94
 Windows GUI applications 94
exception handling 137–138
 .exe file type. *See* file types, .exe
executable files 23, 24, 65, 85
/EXPORT linker command 67, 70
exported names 66–71
 C/C++ name 67, 68, 69, 70
 exporting 66–67
 external name 69, 70–71
 importing 68
 library name 69, 70–71
 linker name 69, 70
extern keyword 68
external names 69, 70–71

F

fabs() 147
File menu
 New Project... command 51
File Name option, x86 Target panel 86
file types
 .dll 23, 66, 73
 .exe 23
 .lib 23, 64, 66
 .mcp 54
 .obj 23
 .rc 75
 .res 75
files
 application. *See* executable
 debugger symbols (SYM) 61, 95, 127, 128
 DLL linker files 66
 executable 23, 24, 65, 85

library 23, 85
linker commands 23, 67, 95
Microsoft formats 24
Portable Executable (PE) format 23
resource 24, 75
resource script 75, 76
types. *See* file types
Win32 API headers 23

float size 104
floating point formats for x86 104, 115
floating-point formats 104
fsqrt() 147
function-calling conventions 24
functions, library. *See* library functions

G

GDI32.LIB library 138
Generate CV Info option, x86 Linker panel 95
Generate CV Info setting 90
Generate Link Map option, x86 Linker panel 95
Generate SYM File option, x86 Linker panel 95
Generate SYM File setting 61, 128
Global Preferences
 Debugger 128

H

header files
 MFCHeaders 139
hook function 73
HTML. *See* Hypertext Markup Language
Hypertext Markup Language (HTML) 15

I

I/O 112
IDE described 19
IDE User Guide 14, 51, 60, 61, 63, 64, 65, 74, 78, 92
inheritance
 multiple 25
 single 24
initialization
 applications 136
 C/C++ run-time library 72, 93, 136
 dynamic link libraries (DLLs) 71–73, 136
inline assembler

- asm blocks 124
- for x86 123–126
- instructions 123, 124
- local variables 126
- syntax 124
- using for x86 123–126
- Inline Intrinsic Functions option, x86 CodeGen
 - panel 90
- Installer, CodeWarrior 132
- installing CodeWarrior 18
- instruction scheduling optimization 108
- Instruction Scheduling option, x86 CodeGen
 - panel 90
- int size 103
- INTEGER 113
 - and arithmetic on x86 113
- integer formats 103
- integer formats for x86 103, 113
- intrinsic functions 147
- IP address 133

K

- KERNEL32.LIB library 138

L

- language extensions for x86 112
- .lib file type. *See* file types, .lib
- libraries
 - ANSICPPX86.LIB 136
 - Ansicx86.lib 136
 - ANSI-standard 135
 - C/C++ run-time 72, 93, 136, 137
 - CPPEXC.DLL 137
 - dynamic. *See* dynamic link libraries (DLLs)
 - GDI32.LIB 138
 - KERNEL32.LIB 138
 - Metrowerks Standard Libraries (MSL). *See* Metrowerks Standard Libraries
 - MFC 135, 139
 - MFC.LIB 139
 - MWCTL.lib 72
 - Mwctl.lib 136, 138
 - Mwctl40.dll 136
 - MWCTL.D.LIB 72
 - Mwcrtd.lib 136

- Mwcrtd40.dll 136
- Mwcrtdll.lib 136
- Mwcrtdlld.lib 136
- Mwcrtdlldmse.lib 136
- Mwcrtdllmse.lib 136
- Mwcrtdmse.lib 136
- Mwcrtdmse40.dll 136
- Mwcrtdmse 136
- Mwcrtdmse40.dll 136
- standard for Win32/x86 135–140
- static. *See* static libraries
- USER32.LIB 138
- version number 95
- Win32 API 135, 138–139

- library files 23, 85

- library functions, MSL 139

- library names 69, 70–71

- link order 109

- linker

- .a files 109

- .o files 109

- described 22

- for x86 108–109

- multiply defined symbols 109

- other documentation 101, 111

- linker. *See* CodeWarrior, x86 linker

- linker command file 23, 67, 95

- linker commands

- /EXPORT 67, 70

- linker names 69, 70

- Linker option, Target Settings panel 83

- Linker setting 57

- linking 27, 60

- See also* IDE User Guide

- local variables in inline assembler 126

- long double size 104

- long long size 103

- long size 103

- LONGINT

- and arithmetic on x86 113

- LONGREAL 116

M

- Machine Code Listing option, x86 CodeGen
 - panel 92

Index

- MacTCP 133
- `_main` function 94
- `_mainCRTStartup` entry-point function 94
- Make** command 60
- makefiles 26
- mangled names. *See* names, mangled
- `.mcp` file type. *See* file types, `.mcp`
- menu commands
 - Add Files** 136
 - Make** 60
 - New Project** 51
 - Preferences** 132
 - Settings** 55
- menus
 - Edit** 132
 - Edit** (CodeWarrior IDE) 55
 - File** 51
 - Project** 60, 136
- message map 42
 - code example 42
- Metrowerks Standard Libraries (MSL) 135–138, 139
 - exception handling 137–138
 - file and memory restrictions 137
- Metrowerks Standard Libraries *See* MSL
- MFC. *See* development tools, Microsoft Foundation Classes application framework
- MFC.LIB library 139
- MFCHeaders header file 139
- Microsoft 117
 - `__cdecl` calling convention 118
 - `__fastcall` calling convention 118
 - `__stdcall` calling convention 119
- Microsoft Corporation
 - CodeView debugging information 24, 95, 127
 - Common Object File Format (COFF) 22–23, 24, 25
 - compatibility with software 24–25, 77–78
 - compilers 24
 - exported-name conventions 66
 - file formats 24
 - linker 24
 - MFC library 135, 139
 - Microsoft Foundation Classes (MFC) application framework 15, 24
 - Object Linking and Embedding (OLE) programming model 25
 - Portable Executable (PE) file format 23, 24
 - resource compiler 77–78
 - resource files 24
 - Visual C++ 24, 25
 - Visual C++ Developer’s Studio 25
 - Win32 API libraries 135, 138–139
 - Win32 Application Programming Interface (API) 15, 23, 138
 - Win32 Reference* 76, 140
 - Windows graphical user interface (GUI) 94
 - Windows operating system 23
- Microsoft Foundation Class
 - header file 38
 - member functions 43
- Microsoft Foundation Classes (MFC). *See* development tools, Microsoft Foundation Classes application framework
- Microsoft Win32 Reference* 76, 140
- mode switch 147
- mode switching 142
- modifying
 - target settings 79–81
- MSL
 - described 23
 - See also* MSL C Reference, MSL C++ Reference
- MSL. *See* Metrowerks Standard Libraries
- MSL C Reference* 14, 135, 136
- MSL C++ Reference* 14, 135, 136
- multiple inheritance 25
- multiple symbols and linker 109
- MWCTL.lib library 72
- Mwctl.lib library 136, 138
- Mwctl40.dll library 136
- MWCRTLD.lib library 72
- Mwcrtd.lib library 136
- Mwcrtd40.dll library 136
- Mwcrtdll.lib library 136
- Mwcrtdlld.lib library 136
- Mwcrtdlldmse.lib library 136
- Mwcrtdllmse.lib library 136
- Mwcrtdmse.lib library 136
- Mwcrtdmse40.dll library 136
- Mwcrtdmse library 136

Mwcrtlmse40.dll library 136

N

Name new project dialog 53, 54

Name option, x86 Linker panel 94

Name setting (x86 Linker) 72

names

- decorated 69

- exported 66–71

- exporting 66–67

- importing 68

- mangled 24, 70

- transformations 68–71

Network control panel (Windows) 132

New Project command 51

New Project window 51, 53, 74

number formats

- floating-point 104, 115

- for x86 102–104, 113–116

- integers 103, 113

NuMega Technologies, Inc.

- BoundsChecker debugger 25

- SoftICE debugger 25

O

.obj file type. *See* file types, .obj

object libraries 24

Object Linking and Embedding (OLE) programming model 25

object modules 22, 23, 24

OLE. *See* Object Linking and Embedding programming model

optimizing

- for x86 107–108

- instruction scheduling 108

- register coloring 107

Ord4() function 114

Output Directory option, Target Settings panel 84

P

Pascal

- Calling convention 117

Pascal compiler

- described 21

Pascal Compilers Reference 14, 59, 60, 64, 74

Pascal Language Reference 14

PATH environment variable 74, 138

PE. *See* Portable Executable file format

picking a target 82

Port ID 133

Portable Executable (PE) file format 23, 24

Post-Linker option, Target Settings panel 84

PowerPC

- INTEGER 113

pragmas

- AMD K6 146

- MMX 146

pragmas (C/C++)

- sym 128

preference settings

- Automatically launch applications**

- when SYM file opened** 128

- Connection** (debugger) 133

Preferences command 132

Preferences window 132

Prefix File option, Windows RC panel 87

Pre-Linker option, Target Settings panel 84

preprocessing 28

- See also* IDE User Guide

Program Arguments 129

Program Arguments option, Debugger Runtime Settings panel 97

Project menu

- Add Files** command 136

- Make** command 60

project stationery 51, 53, 74, 80, 139

Project Type option, x86 Target panel 85

Project Type setting 58, 64, 73

project window 54, 64, 128

projects

- building 60

- compiling 60

- creating 51–54

- debugging 61

- linking 60

R

Ray tracing 29

Index

RC. *See* CodeWarrior, Win32 resource compiler
REAL 115
register coloring optimization 107
register variables for x86 106
registers
 variables 106
Remote Debugging 130–133
remote debugging 21, 130, 131–133
 communications nub 21, 130
 configuring host (MacOS) system 132–133
 configuring target (Win32/x86) system 130–133
 network (TCP/IP) connection 131–132
 serial connection 131
Remote IP Address 133
Remove option, Debugger Runtime Settings panel 97
requirements *See* system requirements
Reserved Heap Size option, x86 Target panel 86
Reserved Stack Size option, x86 Target panel 86
resource compiler (RC). *See* CodeWarrior, Win32 resource compiler
resource editor, CodeWarrior Constructor 76
resource files 24, 75
resource scripts 75, 76
 statements 76–77
resources, creating 75–78
run-time library. *See* libraries, C/C++ run-time
Runtime Settings
 debugger 129

S

Save Project Entries, Target Settings panel 84
selecting a target 82
serial connection 21, 130, 131, 133
Settings command 55
settings panels
 Debugger 97–98
 Debugger Runtime 95–97
 for x86 81–99
 selecting 80
 Target Settings panel 82–84
 Windows RC 86–87
 x86 Linker 92–95
 x86 Processor 87–92

 x86 Target 84–86
 See also target settings
 short double size 104
 short size 103
 signed char size 103
SINGLE 116
single inheritance 24
SoftICE debugger (NuMega) 25
Source Model setting 59
static libraries 65, 85
 creating 63–64
 defined 63
 and DLLs 66, 68
 standard for Win32/x86 135
__stdcall declaration specifier 69, 70, 71
SubSystem option, x86 Linker panel 94
SubSystem setting 94
SubSystemID option, x86 Linker panel 94
SYM Format option, x86 CodeGen panel 90
SYM Format setting 61, 128
sym pragma 128
symbol (SYM) file, debugger 61, 95, 127, 128
symbols
 multiple linker 109
system requirements 17–18

T

Target Name option, Target Settings panel 83
Target Processor option, x86 CodeGen panel 88
target settings 79–99
 Command File 67
 debugging 129
 dialog box 79
 Entry Point Usage 72, 74
 Generate CV Info 90
 Generate SYM File 61, 128
 Linker 57
 modifying 79–81
 Name (x86 Linker) 72
 overview 79–81
 Project Type 58, 64, 73
 Source Model 59
 SubSystem 94
 SYM Format 61, 128
 See also settings panels

Target Settings dialog box 79
Target settings dialog box 56
Target Settings panel 57, 82–84
target settings panels
 Access Paths 78
 C/C++ Language 59
 Target Settings 57
 x86 CodeGen 95, 128
 x86 Linker 61, 67, 72, 74, 90, 127, 128
 x86 Processor 61, 127
 x86 Target 58, 60, 64, 73
target settings window 55
target, selecting a 82
TCP/IP 133
TCP/IP control panel (Mac OS) 132
TCP/IP network protocol 21, 130, 131–132, 133
Turbo Pascal 112
Tutorial
 CodeWarrior 29–50
 Win32 29
Tutorials, CodeWarrior 15
two-machine debugger 130

U

unsigned char size 103
unsigned int size 103
unsigned long long size 103
unsigned long size 103
unsigned short size 103
UNSIGNEDBYTE 114
UNSIGNEDLONG 115
UNSIGNEDWORD 115
USER32.LIB library 138
UserID option, x86 Linker panel 95
USES clause 122
Value option, Debugger Runtime Settings panel 97
variable allocation for x86 106
Variable option, Debugger Runtime Settings panel 97
variables
 register 106
vectorization 143
version numbers
 applications 95
 libraries 95
Visual C++ (Microsoft) 24, 25
 Developer's Studio 25
Win32 Application Programming Interface (API) 15, 23, 138
 software development libraries 135, 138–139
Win32 resource compiler (RC). *See* CodeWarrior, Win32 resource compiler
Win32/x86
 registers 106
windows
 New Project 51, 53, 74
 Preferences 132
 project 54, 64, 128
 target settings 55
Windows 95/NT
 Calling conventions
 __cdecl 118
 __fastcall 118
 __stdcall 119
 CodeWarrior default 117
 overview 116
 overview of Microsoft-standard 117
 Turbo Pascal I/O support 112
Windows operating system 23
 graphical user interface (GUI) 94
Windows RC panel 86–87
 _winMain function 94
 _winMainCRTStartup entry-point function 94
Working Directory 129
Working Directory option, Debugger Runtime Settings panel 97
 _wWinMain function 94
x86 CodeGen panel 95, 128
x86 linker. *See* CodeWarrior, x86 linker
x86 Linker panel 61, 67, 72, 74, 90, 127, 128
x86 Processor panel 61, 127
x86 Target panel 58, 60, 64, 73
x86
 DOUBLE 116
 LONGREAL 116
 REAL 115
 SINGLE 116

Index

UNSIGNEDBYTE 114	x86 Settings
UNSIGNEDLONG 115	Port ID 133
UNSIGNEDWORD 115	x86 settings panels <i>See</i> settings panels
x86 Linker panel 92–95	x86 Target panel 84–86
x86 Processor panel 87–92	

CodeWarrior

Targeting Win32

Credits

writing lead:	Alisa Dean
other writers:	Carl B. Constantine, Steve Chernicoff, Sarah Markey, and Jim Trudeau
engineering:	Bob Kushlis, Rajeev Gulati, Ben Combee, Kurt Ostfeld, Eric Cloninger, Nalini Prakash, Joel Sumner, Cam Vien, and HongGang Zhang
frontline warriors:	Derek Saldana, David Blache, Frank Tur- ovich, Marc Paquette, John Roseborough, and CodeWarrior users everywhere



Guide to CodeWarrior Documentation

CodeWarrior documentation is modular, like the underlying tools. There are manuals for the core tools, languages, libraries, and targets. The exact documentation provided with any CodeWarrior product is tailored to the tools included with the product. Your product will not have every manual listed here. However, you will probably have additional manuals (not listed here) for utilities or other software specific to your product.

Core Documentation	
IDE User Guide	How to use the CodeWarrior IDE
Debugger User Guide	How to use the CodeWarrior debugger
CodeWarrior Core Tutorials	Step-by-step introduction to IDE components
Language/Compiler Documentation	
C Compilers Reference	Information on the C/C++ front-end compiler
Pascal Compilers Reference	Information on the Pascal front-end compiler
Error Reference	Comprehensive list of compiler/linker error messages, with many fixes
Pascal Language Reference	The Metrowerks implementation of ANS Pascal
Assembler Guide	Stand-alone assembler syntax
Command-Line Tools Reference	Command-line options for Mac OS and Be compilers
Plugin API Manual	The CodeWarrior plugin compiler/linker API
Library Documentation	
MSL C Reference	Function reference for the Metrowerks ANSI standard C library
MSL C++ Reference	Function reference for the Metrowerks ANSI standard C++ library
Pascal Library Reference	Function reference for the Metrowerks ANS Pascal library
MFC Reference	Reference for the Microsoft Foundation Classes for Win32
Win32 SDK Reference	Microsoft's Reference for the Win32 API
The PowerPlant Book	Introductory guide to the Metrowerks application framework for Mac OS
PowerPlant Advanced Topics	Advanced topics in PowerPlant programming for Mac OS
Targeting Manuals	
Targeting BeOS	How to use CodeWarrior to program for BeOS
Targeting Java VM	How to use CodeWarrior to program for the Java Virtual Machine
Targeting Mac OS	How to use CodeWarrior to program for Mac OS
Targeting MIPS	How to use CodeWarrior to program for MIPS embedded processors
Targeting NEC V810/830	How to use CodeWarrior to program for NEC V810/830 processors
Targeting Net Yaroze	How to use CodeWarrior to program for Net Yaroze game console
Targeting Nucleus	How to use CodeWarrior to program for the Nucleus RTOS
Targeting OS-9	How to use CodeWarrior to program for the OS-9 RTOS
Targeting Palm OS	How to use CodeWarrior to program for PalmPilot
Targeting PlayStation OS	How to use CodeWarrior to program for the PlayStation game console
Targeting PowerPC Embedded Systems	How to use CodeWarrior to program for PPC embedded processors
Targeting VxWorks	How to use CodeWarrior to program for the VxWorks RTOS
Targeting Win32	How to use CodeWarrior to program for Windows